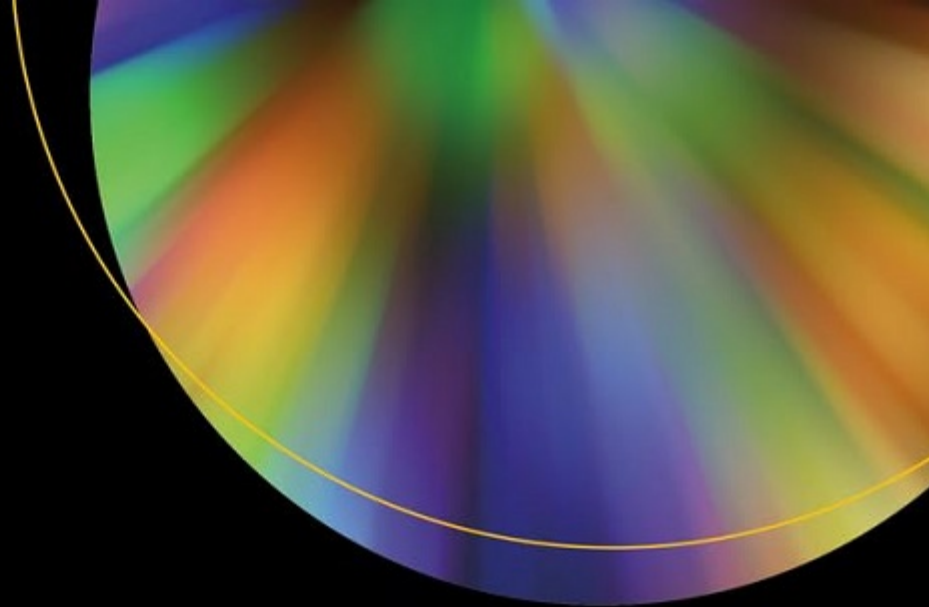




PostgreSQL Query Optimization

The Ultimate Guide to Building
Efficient Queries

—
Second Edition
—

Henrietta Dombrovskaya
Boris Novikov
Anna Bailliekova

Apress®

PostgreSQL Query Optimization

**The Ultimate Guide to Building
Efficient Queries**

Second Edition

**Henrietta Dombrovskaya
Boris Novikov
Anna Baillieкова**

Apress®

PostgreSQL Query Optimization: The Ultimate Guide to Building Efficient Queries, Second Edition

Henrietta Dombrovskaya
DRW Holdings, Chicago, IL, USA

Boris Novikov
Database expert, Helsinki, Finland

Anna Bailliekova
UrbanFootprint, Milwaukee, Wisconsin, USA

ISBN-13 (pbk): 979-8-8688-0068-9

ISBN-13 (electronic): 979-8-8688-0069-6

<https://doi.org/10.1007/979-8-8688-0069-6>

Copyright © 2024 by Henrietta Dombrovskaya, Boris Novikov, Anna Bailliekova

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Shaul Elson
Development Editor: Laura Berendson
Project Manager: Jessica Vakili

Cover designed by Straive

Distributed to the book trade worldwide by Springer Science+Business Media New York, 1 New York Plaza, Suite 4600, New York, NY 10004-1562, USA. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at <http://www.apress.com/bulk-sales>.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub. For more detailed information, please visit <https://www.apress.com/gp/services/source-code>.

Paper in this product is recyclable

Table of Contents

About the Authors.....	xiii
About the Technical Reviewer	xv
Acknowledgments	xvii
Introduction	xix
Chapter 1: Why Optimize?	1
What Do We Mean by Optimization?	1
Why It Is Difficult: Imperative and Declarative	2
Optimization Goals	6
Optimizing Processes.....	7
Optimizing OLTP and OLAP	8
Database Design and Performance	9
Application Development and Performance	11
Other Stages of the Lifecycle	11
PostgreSQL Specifics.....	12
Summary.....	13
Chapter 2: Theory: Yes, We Need It!.....	15
Query Processing Overview	15
Compilation	15
Optimization and Execution	16
Relational, Logical, and Physical Operations.....	17
Relational Operations	17
Logical Operations.....	21
Queries as Expressions: Thinking in Sets	22
Operations and Algorithms	23
Summary.....	23

TABLE OF CONTENTS

Chapter 3: Even More Theory: Algorithms 25

 Algorithm Cost Models 25

 Data Access Algorithms 27

 Storage Structures 27

 Full Scan 28

 Index-Based Table Access 29

 Index-Only Scan 31

 Comparing Data Access Algorithms 32

 Index Structures 34

 What Is an Index? 34

 B-Tree Indexes 36

 Why Are B-Trees Used So Often? 38

 Other Kinds of Indexes 39

 Combining Relations 39

 Nested Loops 40

 Hash-Based Algorithms 42

 Sort-Merge Algorithm 43

 Comparing Algorithms 45

 Summary 45

Chapter 4: Understanding Execution Plans 47

 Putting Everything Together: How an Optimizer Builds an Execution Plan 47

 Reading Execution Plans 48

 Understanding Execution Plans 53

 What Is Going On During Optimization? 54

 Why Are There So Many Execution Plans to Choose From? 55

 How Are Execution Costs Calculated? 56

 How Can the Optimizer Be Led Astray? 59

 Summary 60

Chapter 5: Short Queries and Indexes	61
What Makes a Query “Short”?	62
Choosing Selection Criteria.....	64
Index Selectivity	64
Unique Indexes and Constraints	66
Indexes and Non-equal Conditions	70
Indexes and Column Transformations	70
Indexes and the <i>like</i> Operator	75
Using Multiple Indexes.....	77
Compound Indexes.....	78
How Do Compound Indexes Work?	78
Lower Selectivity	80
Using Indexes for Data Retrieval	81
Covering Indexes	82
Excessive Selection Criteria.....	83
Partial Indexes	89
Indexes and Join Order	90
When Are Indexes Not Used	94
Avoiding Index Usage	94
Why Does PostgreSQL Ignore My Index?.....	94
Let PostgreSQL Do Its Job!	97
How to Build the Right Index(es).....	101
To Build or Not to Build	101
Which Indexes Are Needed?	102
Which Indexes Are Not Needed?	102
Indexes and Short Query Scalability	103
Summary.....	103

TABLE OF CONTENTS

Chapter 6: Long Queries and Full Scans 105

 Which Queries Are Considered Long? 105

 Long Queries and Full Scans..... 107

 Long Queries and Hash Joins..... 108

 Long Queries and the Order of Joins..... 109

 What Is a Semi-join? 109

 Semi-joins and Join Order 111

 More on Join Order 113

 What Is an Anti-join? 116

 Semi- and Anti-joins Using the JOIN Operator 118

 When Is It Necessary to Specify Join Order?..... 120

 Grouping: Filter First, Group Last 123

 Grouping: Group First, Select Last..... 131

 Using SET Operations..... 134

 Avoiding Multiple Scans..... 138

 Conclusion 145

Chapter 7: Long Queries: Additional Techniques 147

 Structuring Queries 147

 Temporary Tables and CTEs 148

 Temporary Tables 148

 Common Table Expressions (CTEs)..... 150

 Views: To Use or Not to Use..... 155

 Why Use Views? 162

 Materialized Views 162

 Creating and Using Materialized Views 162

 Refreshing Materialized Views 164

 Should I Create a Materialized View? 164

 Do Materialized Views Need to Be Optimized?..... 167

 Dependencies..... 167

 Partitioning..... 168

Does Partitioning Improve Performance?	170
Why Create a Partitioned Table?	173
Parallelism	173
Summary.....	174
Chapter 8: Optimizing Data Modification.....	175
What Is DML?	175
Two Ways to Optimize Data Modification	175
How Does DML Work?	176
Low-Level Input/Output.....	176
The Impact of Concurrency Control	177
Data Modification and Indexes.....	180
DML and Vacuum	181
Mass UPDATE/DELETE	181
Frequent Updates	182
Referential Integrity and Triggers.....	182
Summary.....	183
Chapter 9: Design Matters	185
Design Matters.....	185
Why Use a Relational Model?.....	189
Types of Databases.....	190
Entity-Attribute-Value Model	190
Key-Value Model.....	191
Hierarchical Model.....	192
Combining the Best of Different Worlds	193
Flexibility vs. Efficiency and Correctness.....	193
Must We Normalize?	195
Use and Misuse of Surrogate Keys	197
Summary.....	203

TABLE OF CONTENTS

Chapter 10: What About Configuration Parameters? 205

PostgreSQL Configuration Parameters Overview 205

Memory Allocation 206

Connections and Sessions 206

Tuning Parameters for Better Performance 208

Are There Better Ways? 213

Other Limitations of Parameter Tuning 216

Conclusion 216

Chapter 11: Application Development and Performance 217

Response Time Matters 217

World Wide Wait 218

Performance Metrics 219

Impedance Mismatch 219

A Road Paved with Good Intentions 220

Application Development Patterns 220

“Shopping List Problem” 222

Interfaces 224

Welcome to the World of ORM 225

In Search of a Better Solution 227

Summary 230

Chapter 12: Functions 231

Function Creation 231

Internal Functions 232

User-Defined Functions 232

Introducing Procedural Language 233

Dollar Quoting 234

Function Parameters and Function Output: Void Functions 235

Function Overloading 235

Function Execution 237

Function Execution Internals 240

Functions and Performance	243
How Using Functions Can Worsen Performance	243
Any Chance Functions Can Improve Performance?.....	245
Functions and User-Defined Types.....	246
User-Defined Data Types	246
Functions Returning Composite Types	247
Using Composite Types with Nested Structure	250
Functions and Type Dependencies.....	255
Data Manipulation with Functions	255
Functions and Security	257
What About Business Logic?.....	258
Functions in OLAP Systems	259
Parameterizing	260
No Explicit Dependency on Tables and Views.....	261
Ability to Execute Dynamic SQL.....	261
Stored Procedures	261
Functions with No Results.....	261
Functions and Stored Procedures	262
Transaction Management.....	262
Exception Processing	263
Summary.....	264
Chapter 13: Dynamic SQL	265
What Is Dynamic SQL.....	265
Why Dynamic SQL Works Better in Postgres	265
What About SQL Injection?	266
How to Use Dynamic SQL for an Optimal Execution Plan	266
How to Use Dynamic SQL in OLAP Systems.....	274
Using Dynamic SQL for Flexibility	277
Using Dynamic SQL to Aid the Optimizer	286
FDWs and Dynamic SQL.....	289
Summary.....	290

Chapter 14: Avoiding the Pitfalls of Object-Relational Mapping 291

 Why Application Developers Like NORM 292

 ORM vs. NORM 292

 NORM Explained 294

 NORM in the Application Perspective..... 301

 NORM from a Database Perspective 304

 Mapping JSON to the Database..... 305

 Generating Database Code 306

 Getting Data from the Database 308

 Modifying Data in the Database 311

 Why Not Store JSON?! 314

 Performance Gains..... 314

 Working Together with Application Developers..... 315

 Summary..... 315

Chapter 15: More Complex Filtering and Search 317

 Full Text Search..... 317

 Multidimensional and Spatial Search 319

 Generalized Index Types in PostgreSQL 320

 GIST Indexes 320

 Indexes for Full Text Search..... 320

 Indexing Very Large Tables 322

 Indexing JSON and JSONB..... 323

 Summary..... 326

Chapter 16: Ultimate Optimization Algorithm..... 327

 Major Steps..... 327

 Step-by-Step Guide..... 328

 Step 1: Short or Long?..... 328

 Step 2: Short..... 328

Step 3: Long.....	330
Step 4: Incremental Updates	330
Step 5: Non-incremental Long Query	330
But Wait—There Is More!	331
Summary.....	332
Chapter 17: Conclusion.....	333
Index.....	335

About the Authors



Henrietta Dombrovskaya is a database researcher and developer with over 40 years of academic and industrial experience. She holds a PhD in computer science from the University of Saint Petersburg, Russia. At present, she is a database architect at DRW Holdings, Chicago, Illinois. She is an active member of the PostgreSQL community, a frequent speaker at the PostgreSQL conference, and the local organizer of the Chicago PostgreSQL User Group. Her research interests are tightly coupled with practice and are focused on developing efficient interactions between applications and databases.



Boris Novikov's experience includes leading postgraduate research groups and advising dozens of PhD students while partnering and collaborating with industrial clients. His research interests are in the broad area of information management and include design, development, and tuning of databases, applications, and database management systems (DBMSs). He also has interests in distributed scalable systems for stream processing and analytics.

ABOUT THE AUTHORS



Anna Bailliekova is Senior Data Engineer at UrbanFootprint, where she works on data platform tools and cloud ops for geospatial applications. Previously, she worked in data engineering at Zendesk and held a variety of DBA and BI roles at Epic. Her writing can be found in the newsletter *Better Streets MKE*. She received her undergraduate degree with college honors in political science and computer science from Knox College in Galesburg, Illinois.

About the Technical Reviewer

Tom Kincaid is Vice President of Database Development at EnterpriseDB. Tom has been developing, deploying, and supporting database systems and enterprise software for over 25 years. Prior to joining EnterpriseDB, Tom was General Manager of 2ndQuadrant in North America where he oversaw all aspects of 2ndQuadrant's dynamic and growing business for Postgres products, training, support, and professional services. He worked directly with companies from all industries and of all sizes helping them successfully make Postgres part of their mission-critical operations.

Tom has overseen the design and delivery of Postgres training solutions as well as the deployment of PostgreSQL both at Fortune 500 financial institutions and at military facilities all over the world. Teams Tom has managed have delivered major features that have become part of the PostgreSQL open source database.

Tom is also the founder and one of the organizers of the Boston PostgreSQL User Group.

Acknowledgments

It takes many people to bring a book into the world, most of whose names do not appear on the cover. Firstly, we want to thank Jonathan Gennick, who came up with the idea of this book and navigated it through the first edition. Without his initiative, this book wouldn't exist. We're also grateful to the entire team at Apress who have supported this endeavor across two editions.

The contributions of Tom Kincaid as the technical reviewer cannot be overstated. His careful, thorough, and thoughtful feedback improved the content, organization, and usability of the text. This book is more precise, more understandable, and more comprehensive, thanks to Tom. We're grateful that he returned for the second edition, graciously taking another close read. Any remaining issues are, of course, our own responsibility.

After the first edition of this book was released, we heard from many people regarding issues in the `postgres_air` database, topics readers wished were covered, and passages that could have been more clear. This second edition incorporates many of their suggestions. We are grateful to everyone who took the time to read the book closely and share their comments and suggestions with us. In particular, Hannu Krosing provided thorough, detailed, and specific feedback on `postgres_air`, and Egor Rogov provided many helpful suggestions for making the book more understandable and clear.

—Henrietta Dombrovskaya, Boris Novikov, Anna Bailliekova

Thank you to Jeff Czaplewski, Alyssa Ritchie, and Greg Nelson, who spent hours, days, and weeks making No-ORM (NORM) work with Java. My time at EDB was a chance to work with and learn from the best of the Postgres best. My colleagues at DRW—both application and database administrator (DBA) teams—have given me new opportunities to push the limits of Postgres.

—Henrietta Dombrovskaya

ACKNOWLEDGMENTS

I'd like to thank Andy Civettini for teaching me how to write and talk about technical topics in an accessible way and for years of academic and professional encouragement. My colleagues at UrbanFootprint challenge and inspire me every day. Finally, John, Nadia, and Kira Bailliekova have each supported me and sacrificed for the sake of this book; I am endlessly grateful to them.

—Anna Bailliekova

Introduction

“Optimization” is a broad enough term to encompass performance tuning, personal improvement, and marketing via social engine and invariably evinces high hopes and expectations from readers. As such, it is prudent to begin not by introducing what is covered, but rather, why this book exists and what will not be covered, to avoid disappointing readers who approach it with inappropriate expectations. Then, we proceed with what this book is about, the target audience, what is covered, and how to get the most use out of it.

Why We Wrote This Book

Like many authors, we wrote this book because we felt we could not *not* write it. We are educators and practitioners; as such, we see both how and what computer science students are taught in class and what knowledge they lack when they enter the workforce. We do not like what we see and hope this book will help bridge this gap.

When learning about data management, most students never see a real production database, and even more alarming, many of their professors never see one, either. While lack of exposure to real-life systems affects all computer science students, the education of future database developers and database administrators (DBAs) suffers the most. Using a small training database, students can learn how to write syntactically correct SQL and perhaps even write a `SELECT` statement that accurately retrieves desired data. However, learning to write performant queries requires a production-sized dataset. Moreover, it might not be evident that performance might present a problem if a student is working with a dataset that can easily fit into the computer’s main memory and return a result in milliseconds regardless of the complexity of the query.

In addition to lacking exposure to realistic datasets, students often don’t use DBMSs that are widely used in industry. While the preceding statement is true in relation to many DBMSs, in the case of PostgreSQL, it is even more frustrating. PostgreSQL originated in an academic environment and is maintained as an open source project, making it an ideal database for teaching relational theory and demonstrating database internals. However, so far, few academic institutions have adopted PostgreSQL for their educational needs.

As PostgreSQL is rapidly developing and becoming a more powerful tool, more and more businesses are choosing it over proprietary DBMSs, in part to reduce costs. More and more IT managers are looking for employees who are familiar with PostgreSQL. More and more potential candidates learn to use PostgreSQL on their own and miss opportunities to get the most out of it.

We hope that this book will help all interested parties: candidates, hiring managers, database developers, and organizations that are switching to PostgreSQL for their data needs.

What Won't Be Covered

Often, when users start to complain that “everything is slow” or nightly backups finish in broad daylight and everyone seems to be saying that “the database needs to be optimized,” conversations are almost exclusively focused on database configuration parameters and occasionally on underlying Linux parameters. It is often assumed that as soon as the correct values for parameters are chosen and the database instance has been restarted, all the world's problems will be solved.

Countless times, we have worked with customers expecting magical incantations and cheat codes. Countless times, these customers have expressed a deep disappointment when the “wizards” to whom they have appealed suggest looking for the queries that are executed most often or check for missing indexes. After listening politely to all our suggestions, they kept asking: So can you suggest *any other parameter changes*?

Indeed, it is tempting to be able to solve every problem at once. That temptation leads to the popular belief that there is a sort of dark magic and secret codes, a button hidden under a desk somewhere that makes the database run faster.

Since we are aware of these misconceptions, we want to be transparent from the very beginning. The following is the list of topics that are often discussed in books about optimization but will not be covered in this book and why:

- *Server optimization* – With the mass migration to various cloud environments and existing organizational structures, database developers are unlikely to have a say in how servers are configured.

- *PostgreSQL configuration parameters* – In this second edition of the book, we do cover this topic. However, it comprises a relatively small portion of the book, as their impact on performance is overrated, as we will demonstrate, and usually, database developers do not have the necessary privileges to alter them (with a few exceptions).
- *Distributed systems* – We do not have enough industrial experience with them.
- *Transactions* – Their impact on performance is very limited (although we will cover some cases when they can have a major impact).
- *New and cool features* – These change with every new release, and our goal is to cover the fundamentals.
- *Magic, rituals, cheat codes, etc.* – We are not proficient in these approaches to optimization.

There are plenty of books available that cover all of the topics listed previously, except the last, but this book is not one of them. Instead, we focus on everyday challenges database developers face: when that one application page keeps timing out, when a customer is kicked out of the application just before the “Contract Signed” page, when the CEO dashboard is showing an hourglass instead of yesterday’s product KPI, or when procuring more hardware is not an option.

Everything we are presenting in this book has been tested and implemented in an industrial environment, and though it may look like magic, we will explain any query performance improvement or lack thereof.

Target Audience

Most of the time, a book about optimization is viewed as a book for DBAs. Since our goal is to prove that optimization is more than just building indexes, we hope that this book will be beneficial for a broader audience.

This book is for IT professionals working in PostgreSQL who want to develop performant and scalable applications. It is for anyone whose job title contains the words “database developer” or “database administrator” or who is a backend developer charged with programming database calls. It is also useful to system architects involved in the overall design of application systems running against a PostgreSQL database.

What about report writers and business intelligence specialists? Unfortunately, large analytical reports are most often thought of as being slow by definition. However, if a report is written without considering how it will perform, the execution time might end up being not just minutes or hours, but years! For most analytical reports, execution time can be significantly reduced by using simple techniques covered in this book.

What Readers Will Learn

In this book, the readers will learn how to

- Identify optimization goals in OLTP (Online Transaction Processing) and OLAP (Online Analytical Processing) systems
- Read and understand PostgreSQL execution plans
- Identify indexes that will improve query performance
- Optimize full table scans
- Distinguish between long queries and short queries
- Choose the right optimization technique for each query type
- Avoid the pitfalls of ORM frameworks

At the end of the book, we present the *Ultimate Optimization Algorithm*, which guides a database developer through the process of producing the most performant query.

The Postgres Air Database

Throughout this book, examples are built on one of the databases of a virtual airline company called Postgres Air. This company connects over 600 virtual destinations worldwide, offers about 32,000 direct virtual flights weekly, and has over 100,000 virtual members in its frequent flyer program and many more passengers every week. The company fleet consists of virtual aircraft.

Please note that all data provided in this database is fictional and provided for illustrative purposes only. Although some data appears very realistic (especially descriptions of airports and aircraft), they cannot be used as sources of information about real airports or aircraft. All phone numbers, email addresses, and names are generated.

To install the training database on your local system, please refer to the GitHub repo: github.com/Hettie-d/postgres_air.

The README.md file contains the link to the data directory and detailed installation instructions.

In addition, after you restore the data, you will need to run the script in Listing 1 to create several indexes.

Listing 1. Initial set of indexes

```
SET search_path TO postgres_air;
CREATE INDEX flight_departure_airport ON flight(departure_airport);
CREATE INDEX flight_scheduled_departure ON flight (scheduled_departure);
CREATE INDEX flight_update_ts ON flight (update_ts);
CREATE INDEX booking_leg_booking_id ON booking_leg (booking_id);
CREATE INDEX booking_leg_update_ts ON booking_leg (update_ts);
CREATE INDEX account_last_name ON account (last_name);
```

We will use this database schema to illustrate the concepts and methods that are covered in this book. You can also use this schema to practice optimization techniques.

This schema contains data that might be stored in an airline booking system. We assume that you have booked a flight online, at least once, so the data structure should be easily understood. Of course, the structure of this database is much simpler than the structure of any real database of this kind.

Anyone who books a flight needs to create an account, which stores login information, first and last names, and contact information. We also store data about frequent flyers, which might or might not be attached to an account. A person who makes a booking can book for several passengers, who might or might not have their accounts in the system. Each booking may include several flights (legs). Before the flight, each traveler is issued a boarding pass with a seat number.

The Entity-Relationship (ER) diagram for this database is presented in Figure 1.

- *airport* stores information about airports and contains the airport's three-character (IATA) code, name, city, geographical location, and time zone.

INTRODUCTION

- *flight* stores information about flights between airports. For each flight, the table stores a flight number, arrival and departure airports, scheduled and actual arrival and departure times, aircraft code, and flight status.
- *account* stores login credentials, the account holder's first and last names, and possibly a reference to a frequent flyer program membership; each account may potentially have multiple phone numbers, which are stored in the *phone* table.
- *frequent_flyer* stores information about membership in the frequent flyer program.
- *booking* contains information about booked trips; each trip may have several booking legs and several passengers.
- *booking_leg* stores individual legs of bookings.
- *passenger* stores information about passengers, linked to each booking. Note that a passenger ID is unique to a single booking; for any other booking, the same person will have a different passenger ID.
- *aircraft* provides the aircraft's description, and the *seat* table stores seat maps for each of the aircraft types.
- Finally, the *boarding_pass* table stores information about issued boarding passes.

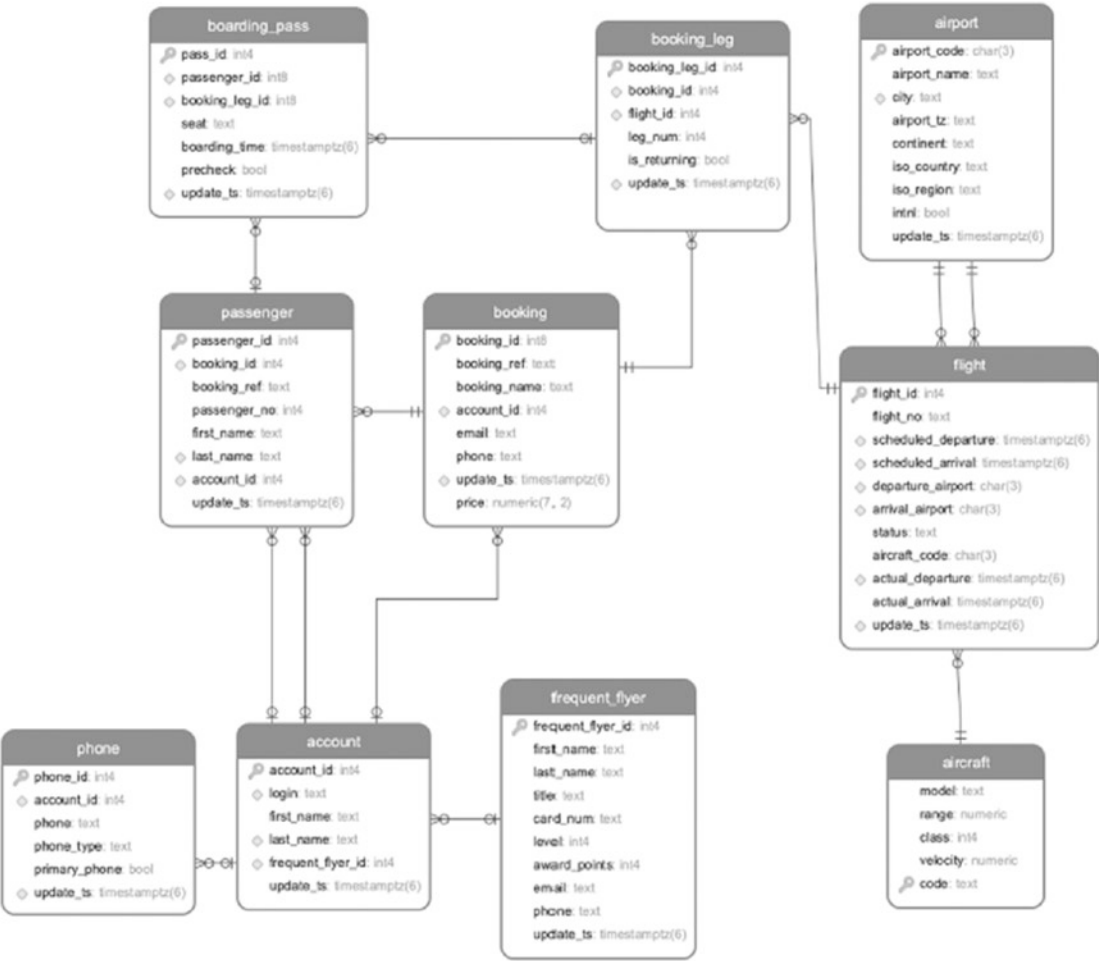


Figure 1. ER diagram of the booking schema

The Key to Digital Transformation Success

In the age of digitization, businesses face a critical imperative: to adapt and embrace innovation or risk being left behind in a rapidly evolving world.

The concept of creative destruction, which is the process where innovation and technology advancements are reshaping industries and business models, is the driving force behind digital transformations. In this chapter, you'll explore what digital transformation is and the state of digital transformation programs where unfortunately there is a high failure rate. Learn about the challenges of digital transformation and the key role strategic alignment plays in its success. Discover how process serves as a common language, aligning organizations vertically and horizontally, and explore the Process Inventory

framework's role in identifying, managing, and aligning processes. By the end of this chapter, you'll be equipped with an understanding of how Process Inventory not only enables digital transformation's success but also has the power to transform how the organization integrates technical capabilities, leads to operational excellence, streamlines the management of change, and enhances risk management capabilities.

Nokia's Mobile Phone Business

Founded in 1865 in Finland, Nokia initially focused on various industries, including pulp and paper, rubber, and electronics [1]. In the 1990s, Nokia made a strategic decision to shift its focus toward mobile phones. The company overtook Motorola in October 1998 to become the best-selling mobile phone brand in the world. Their net operating profits went from \$1 billion in 1995 to almost \$4 billion by 1999. A major reason for their success was catering to a youthful consumer market and fashion-oriented consumers with colorful and replaceable back covers. They were also innovators, introducing the first camera phone to the market in 2002 and pioneering mobile gaming with a preloaded version of Snake.

In 2007, Apple introduced the iPhone. By Q4 2007, half of all smartphones sold in the world were Nokias, while the iPhone market share was 5% [2].

The iPhone revolutionized the mobile industry with its touchscreen interface and robust app ecosystem. Nokia underestimated the impact of smartphones and the changing preferences of consumers. They continued to rely on their traditional strengths, such as hardware design and the Symbian operating system, while underestimating the importance of software and user experience. The Symbian platform quickly became outdated and posed difficulties for developers after the advent of iOS and Android. Nokia's market share dropped by 90%, and in 2013, their mobile phone business was acquired by Microsoft.

A study named "Distributed Attention and Shared Emotions in the Innovation Process: How Nokia Lost the Smartphone Battle" [3] was conducted, interviewing 76 of Nokia's top and middle managers, engineers, and external experts to determine the root causes. The study found that the downfall can be attributed to a lack of organizational alignment. It highlighted several factors, including a culture of fear, intimidated middle managers, a lack of technical competence among top managers, and a short-term focus on market demands. The internal politics and shared fear created a climate of mistrust that hindered innovation.

Creative Destruction in the Age of Digitization

The concept of creative destruction, introduced by Austrian American economist Joseph Schumpeter in 1942 [4], has impacted various companies, such as Nokia, Blockbuster Video, Kodak, Blackberry, and Toys 'R' Us to name a few. Creative destruction describes the process whereby innovation and technological advancements disrupt established industries and business models.

According to Schumpeter's theory, it is inherent in human nature to be creative and strive for an improved standard of living. Additionally, the pursuit of profit incentives drives constant innovation and technological advancements [5]. In a competitive environment, producers generate ideas that advance technologies, redesign existing products and processes, and introduce new products. This competition results in improved products at lower costs, ultimately benefiting consumers.

This continuous process of innovation and economic competition contributes to overall economic growth and enhances the standard of living for consumers. However, it also poses a significant risk to incumbent organizations that fail to adapt as they can be displaced or even eliminated by more innovative competitors. This is driving the urgency in many organizations to invest in their own digital transformation program.

Digital Transformation Defined

Digital transformation is the application of digital capabilities to processes, products, and assets to improve efficiency, enhance customer value, manage risk, and uncover new monetization opportunities [6]. There are many digital capabilities driving this, such as artificial intelligence, cloud computing, Internet of Things (IoT), big data analytics, blockchain, and more. This goes beyond simply acquiring these new technologies and teaching your staff to use them.

Digital transformation must be framed in a business context, which is why the preceding definition includes "uncover new monetization opportunities." Monetization opportunities may mean new product revenue streams, deepening customer relationships, expanding to new ones, or reducing the cost of doing business. The implications of maximizing these capabilities for an organization can be far-reaching and may fundamentally change core business models.

The essence of digital transformation lies in becoming a customer-centric, data-driven, and agile organization. It entails ensuring that key decisions, actions, and processes are guided by data-driven insights rather than just human intuition [7].

The challenge, and opportunity, is that the capabilities of these technologies are advancing at a rapid rate, making it hard to accurately predict how they will impact businesses in the next five years or beyond. The external environment must also be factored in, such as changes to the competitive landscape, customer preferences, globalization, and regulatory environment.

This means that digital transformation must be viewed as a continuous journey, requiring a strong digital strategy and adaptive execution to respond to changes through iterations and continuous learning. A crucial factor in being adaptive is achieving operational excellence, which involves optimizing and maximizing efficiency across all resources leveraged to operate your business. By embracing operational excellence, organizations ensure their flexibility and responsiveness in the face of advancing digital capabilities and evolving external environments.

State of Digital Transformation

According to a study, digital companies achieve 1.8 times higher [8] earnings growth. Technology advancements mean that digital companies have greater productivity improvements, better customer experiences, and stronger innovation. The global digital transformation market is projected to grow from \$695.5 billion in 2023 to over \$3.14 trillion by 2030, at a compound annual growth rate (CAGR) of 24.1% during this period [9]. About 89% [10] of enterprises are planning to adopt or have already adopted a digital business strategy.

Shockingly, 70% of such initiatives have been found to fail to reach their goals [8]. This represents a massive investment with a very high failure rate. The implication is more than just lost investment; it also means frustrated stakeholders, lost ground to competitors, unsatisfied customers, and in some cases, costs such as high regulatory penalties.

Why Digital Transformations Fail

According to McKinsey [11], the reasons transformations fail are as follows:

- **Lack of Aspiration:** The CEO fails to set a sufficiently high aspiration. People throughout the organization do not buy in, and they lack the motivation to invest extra energy in making the change happen.
- **Lack of a Shared Vision:** During the early stages of the transformation, the CEO does not build conviction within the team about the importance of the change or craft a change narrative that convinces people they need to drive the transformation.

- **Lack of Engagement:** People throughout the organization do not buy in, resulting in a lack of willingness to invest the extra energy required to drive the transformation.
- **Low Investment in Capability Building:** The leadership team neglects to address skill gaps within the organization or fails to free up key resources from their other day-to-day responsibilities.
- **Insufficient Structure of the Transformation Team:** The organization fails to establish the right change management structure and cadence for leadership and transformation management oversight.

For many medium and large organizations, these can be challenging hurdles to overcome due to their complexity. With organizations that have tens or hundreds of thousands of employees, it becomes an immense challenge to align and coordinate all those resources, especially when the transformation goals are ambitious. However, complexity alone cannot be solely blamed as there are many digitally successful organizations that effectively manage complexity.

The underlying root cause of these challenges is a lack of **strategic alignment**. Strategic alignment means that all elements of a business, including market strategy and the way the company is organized, are arranged in such a way as to best support the fulfillment of its long-term purpose [12]. This means aligning the purpose, vision, and mission statements, which articulate why the organization exists and the future they are driving toward, with strategy, which is how the business will achieve that purpose. Strategy defines which products and services to offer, which markets to serve, and in this context, how they will adopt digital technology to further support the purpose of the organization. Strategy needs to be aligned with the capabilities of the organization, which are all resources including people, processes, technology, data, vendors, etc. that support the organization. An example is an organization that is looking to achieve superior customer service must translate that into a high level of excellence when frontline staff are engaging with customers directly.

There are two important aspects of strategic alignment: vertical alignment and horizontal alignment. Vertical alignment refers to aligning strategies, goals, and performance metrics hierarchically from the CEO through middle management down to the single contributors. Horizontal alignment means that business units and functional areas such as technology, finance, and risk collaborate strongly across these areas and avoid siloed operating models, which bring a great number of inefficiencies. Achieving strategic alignment facilitates unification across resources which not only enables an organization to meet their digital transformation goals but enables organizational agility.

But how do organizations achieve strategic alignment? After all, this is not a new concept, yet it exists in so few organizations. I believe the answer lies in language. Everyone in these organizations is focused on their responsibilities, and they develop their perspectives based on their work. This often results in lower-level contributors not fully understanding the strategic messages coming from the CEO. As a result, they may fail to invest their energies in contributing to the transformational ambitions of leadership. When a person from a business unit conveys requirements to a technology professional, they may struggle to accurately communicate what they are looking for, leading to poor requirements, or delivered software that misses the mark. This lack of understanding and effective collaboration contributes to the formation of redundancies within organizations, where different functional units create processes that others already perform.

A common language should create a shared understanding vertically and horizontally. It would align objectives, facilitate better communications, bridge cultural differences, and enable conflict resolution. What are the requirements of this common language? First, since it must support the purpose of the organization, it has to be oriented in the language of the business. This eliminates, for instance, technical terminology, you wouldn't want to state as a strategy that you're going to implement artificial intelligence unless you could convey how it supports your customers, generates revenues, or furthers the mission. But it does have to be a bridge to cross-functional topics such as technology, risk, data, and regulators. Lastly, it needs to connect a high-level view that senior leadership would understand to a low-level view that a single contributor would understand.

Process Is the Common Language

The only candidate that meets those requirements is process. I'll illustrate this in a story that highlights how organizations evolve with process at its core.

Let's say that you are talking to a couple of friends about starting a business. That conversation begins with the actions or processes that must occur. The conversation would go something like this: "We can create great designs, print them on t-shirts, and sell them at concerts." The purpose of the company will be to make money or get a message out, but in that statement, the business processes have been identified:

- Create designs
- Print design on T-shirts
- Sell at concerts

Let's say that this company has success. Then they realize that they will have to file taxes, but none of them have the skill set or the time, so they hire someone for that purpose and have now added the process of

- Calculate tax liabilities
- File taxes
- Pay tax liabilities

This means that they've now expanded the organization chart to support the need to do taxes. Then they purchase tax software for them which is the beginning of their technology portfolio. The business is doing so well that you seek out investors, but investors will want financial reports that are based on solid accounting standards, which leads to them hire an accountant to perform the following processes:

- Close books
- Create financial report

This accountant will require accounting software. Now they've started a finance functional unit which has tax and accounting processes, an organization chart, and a technology portfolio.

I can go on, but what you can see is that the needs of the organization to support its purpose drive the need for processes. Process is core to organizational capabilities as people are hired, technology is purchased or built, and risk is incurred, all to support the processes of the business. Plus, a hierarchy of processes evolves as processes get more complex and lower-level resources are hired to perform more detailed tasks.

The Process Inventory Framework

The Process Inventory is a framework for capturing every process within an organization and a methodology for managing this information sustainably over time. One of the guiding principles for Process Inventory is that it must be a complete representation of processes for the organization in scope. To accomplish this, interviews must be conducted following the organizational chart since that represents a complete inventory of people in the organization.

These interviews start at the top of the organizational chart, working down from business units or functional levels to the team or individual contributor level. Stakeholders are requested to provide formal attestations to ensure completeness and accuracy in the inventory, including any process inventories collected by their teams.

Starting from the top of the organizational chart enables vertical alignment, allowing for a direct trace of process ownership from the CEO to individual contributors at all levels.

Organizations often have multiple sources of information that describe various aspects of their operations, such as application repositories, risk repositories, product, and channel repositories, and more. However, this data is usually managed in silos and lacks concrete associations across topics, hindering horizontal alignment.

The Process Inventory framework establishes a taxonomy that classifies all processes executed to support the organization's purpose. This taxonomy serves as an index for aligning different aspects of the organization through a single business-oriented language. A modeling team can extract information from these repositories and create associations, at the process name level or through creating process models, where operational information intersects with processes.

This achieves horizontal alignment in three ways. Firstly, it enables functional teams to precisely identify how their concepts intersect with processes. For example, by examining the intersection of applications in an application repository with processes, it becomes possible to determine which applications support specific processes and even drill down to the level of identifying which APIs within those applications support specific steps in a process. This is valuable in identifying the scope and requirements for change management initiatives. Secondly, it provides transparency to all stakeholders regarding the activities of each business unit and functional team. This transparency helps break down silos and facilitates better coordination. Lastly, it promotes a culture of accountability by identifying points of ownership and fostering strong collaboration across organizational units where necessary.

This framework is analogous to Google Maps, where you can view Earth from space, but then you can zoom in to see details of your house and the signs on your street. Google's initiative called Ground Truth [13] marries numerous data sets about our world, such as traffic patterns, restaurant ratings, and street view images to physical locations. This has revolutionized not only navigation but also how we interact in the world and how advertisers reach us.

The Digital Transformation Journey

In a digital transformation, this common language enables the leadership team to articulate their aspirations for a digital future, and the transformation teams can identify specific processes that need to change or processes to be added. The shared vision can be cascaded throughout the organization, clearly defining responsibilities and performance metrics to drive the transformation. This culture of accountability will foster engagement as everyone will

understand how their work directly contributes to the bigger picture. Essentially, this enables solutions to the typical failures encountered in digital transformations.

In this book, I will extensively detail the constructs of this framework, including the operating model and the methods to create and maintain the process inventory and associated metadata sourced from various repositories.

I will cover the method of defining a digital strategy and how to set up the transformation program for success. The core of this process involves defining the digital transformation strategy in the context of how it supports the organization's purpose and aligns with operational capabilities. This encompasses defining the business areas and use cases that need improvement, along with the ways of working. I provide details on how the Process Inventory framework can transform these use cases.

A significant aspect of adapting to the digital age is integrating new **technology** capabilities. Technology is typically one of the largest investments an organization makes and can be a source of pain, especially when the technical architecture becomes rigid. I will detail how this framework can lead to much better IT architecture documentation, creating alignment between IT assets and the processes they serve. This visibility can help rationalize complexity by eliminating redundant systems and increasing reuse, which is key to flexibility and efficiency. Additionally, I will demonstrate how this directly supports the design and implementation of digital technologies such as intelligent process automation, the adoption of artificial intelligence and machine learning at scale, and how to build a business-led modular architecture design, which is crucial for an agile architecture.

Operational excellence is achieved with this level of transparency at your fingertips. You can precisely identify all the resources that support a business or process area, enabling the identification of inefficiencies or opportunities to streamline operations. Continuous improvement will be promoted by a framework that enables better monitoring of business processes and the collection of key performance indicators.

Workforces will be transformed by having greater training and development programs to equip staff members with skills and knowledge, not only in their domain but also with a broader understanding. This will foster a culture of innovation by unlocking knowledge that was held by so few and encouraging everyone to identify better ways of doing things.

Change management is a significant challenge for many organizations, particularly in large change efforts. This framework has the potential to greatly transform the entire process. The alignment of strategic priorities can be mapped to the impacted processes. This clarity brings more objectivity to the change portfolio investment process, resulting in stronger effectiveness in the allocation of discretionary funds to projects with the greatest strategic needs.

Project scope can be stated in terms of the business processes involved. This approach ensures certainty of impacts across all resources leveraged by the business process, including people, processes, controls, data, and technology. The result is enhanced cost and timeline estimates and better resource alignment.

The change process would undergo significant changes by anchoring it to this framework. Framing the contents of SDLC, Agile, and organizational change management deliverables, such as business requirements, agile user stories, technology designs, control designs, testing scripts, and user training, using the Process Inventory taxonomy as the structure will increase coordination across teams and enhance traceability for those responsible for project management in these efforts. This also has the benefit of generating better change status reports by communicating the progress and readiness of individual processes as the numerator for percent complete reporting.

The efficiencies provided by this framework will enable faster change implementation with reduced investment needs, thereby freeing up resources to concentrate on enhancing customer experience and driving product innovations.

Risk management is strengthened by giving risk professionals a comprehensive inventory of processes, which will enable them to perform better risk assessments and create a more accurate inventory of risks across all risk types. This allows them, in conjunction with their business partners, to prioritize and design effective controls and ensure regulatory compliance. The risk operating model will be empowered across all three lines of defense for stronger effectiveness and coordination such as arming internal audit with information needed to effectively test controls. This will give senior leaders and boards confidence that they are getting accurate risk information to make their strategic decisions.

Framework Implementation Considerations

This may seem overwhelming, but I assure you it's not. I have dedicated 12 years of my career to developing and testing this framework in various scenarios. I have successfully implemented it at multiple large financial institutions, tailoring it to their specific needs. I have refined the methods to be efficient with limited resources and to ensure the quality of the models, which forms the basis for all the benefits I have presented.

The framework is designed to be incremental and scalable according to your requirements as defined by your transformation strategy. It can be applied at the enterprise level or selectively to specific business units or functions. You have the flexibility to implement only the use cases that are relevant to your organization.

Indeed, adopting this framework requires an investment of time and resources. However, it's important to recognize that organizations pay a price for misalignment. You have a choice: invest upfront to avoid inefficiencies, costly penalties, or losing market share to more digitally adept competitors, or bear the consequences later.

Key Takeaways

- **Creative destruction** is driving the digital age through innovation and technological advancements which are reshaping business models and industries.
- **Digital transformation** is the application of digital capabilities to processes, products, and assets to improve efficiency, enhance customer value, manage risk, and uncover new monetization opportunities.
- **High failure rate** as 70% of such initiatives fail to reach their goals, which means lost investment, frustrated stakeholders, lost ground to competitors, unsatisfied customers, and potentially high regulatory penalties.
- **Lack of strategic alignment** is the underlying root cause of failure which hinders collaboration and organizational focus toward strategic goals.
- **A common language** is needed to facilitate alignment. Process, as captured by the Process Inventory framework, provides the common language.
- **Process Inventory** enables digital transformation success and transforms how the organization operates which leads to operational excellence.

In the next chapter, we will describe the Process Inventory framework, which includes models, modeling characteristics, metadata, and conceptual architecture.

Overview of the Process Inventory Framework

In this chapter, you will explore the components of the Process Inventory framework. Delve into the significance of this framework in establishing an ontology to organize organizational knowledge. Understand the methods for constructing and maintaining a Process Inventory, and explore the conceptual architecture used to integrate metadata from diverse authoritative sources. Additionally, you'll gain insights into related models, such as process models, customer journeys, and value streams. The framework, with all its associated components, is critical for structuring and utilizing knowledge effectively within an organization to deliver transformational value.

The Origins of the Process Inventory Framework

I started my career in 1996 at Andersen Consulting, which later became Accenture. My college degree had a focus on information systems, so I began in a technology practice. My first engagement was with a large bank in Charlotte that was developing a direct banking platform, an emerging concept at that time.

I initially joined a production support team where we handled issues with middleware code. Middleware is an application pattern for distributing messages to appropriate platforms. My role involved triaging issues, identifying root causes within the application code, and developing and testing fixes.

During the first 15 years of my career, I primarily focused on technology. I worked as a C++ and Java developer and a solution architect and led large teams in software delivery.

I transitioned from retail banking to commercial banking without having a strong understanding of these businesses. While I knew about retail banks since I was a customer with a checking account, commercial banking was completely foreign to me. My focus was on delivering quality software on time and supporting production to minimize customer impact. Most of my interactions were with other technologists; I was curious about what the business did, but I didn't have a clear path to explore it.

In 2011, I joined EY, which introduced me to a new world. At that time, EY had limited software development engagement, and their clients were typically senior leaders across various business areas such as risk, technology, and more. The work involved helping these leaders develop solutions to enhance their organizations' effectiveness. It was a refreshing shift from my technology-centric mindset.

One of my early engagements at EY was with a large retail bank. They had surplus budgeted funds that needed to be spent; otherwise, the funds would be lost the following year. One of their requirements was to build a business capability model for their retail banking business. While not an exhaustive exercise, we interviewed key individuals in their businesses and created a model with three levels of depth. It was a moment of epiphany for me. The entire business's activities were encapsulated in a single model, exactly what I had been seeking during my technology days to understand what the business did. I was enthused about the potential of this concept to bring diverse perspectives together and what this could mean for delivery on a wide range of topics. From that point on, I positioned myself as a business architect, eager to delve deeper into this concept, and using this branding helped me connect with clients implementing similar approaches.

BUSINESS ARCHITECTURE

Business architecture emerged as a concept in the 1980s. British management consultant Edwin E. Tozier was one of the first to mention the concept in an article he wrote titled “Developing Strategies for Management Information Systems” [1]. In this article, he distinguished between information architecture and business architecture, stating his theory that each business function may be supported by one or more information systems and that information systems need to be adaptable to organizational strategies.

Business architecture has evolved significantly since then. Several standard-setting organizations such as TOGAF (The Open Group Architecture Framework), OMG (Object Management Group), the Business Architecture Guild, and others have been advancing the concepts and practices, although a common definition remains elusive.

The definition that has resonated the most with me is from OMG, which states that business architecture is “a blueprint of the enterprise that provides a common understanding of the organization and is used to align strategic objectives and tactical demands” [1]. This definition always made intuitive sense to me as changes to a physical building wouldn’t be made without such a blueprint, but many organizations will spend significant amounts of money without any similar concept.

The core deliverable in business architecture is the business capability model. A business capability model represents the required capabilities that an organization must deliver for its purpose. They are largely independent of specific organizational structures, reporting relationships, political agendas, cultural aspects of individual business leaders, current initiatives, and projects [2]. Business capability models have been used for various purposes, such as strategic planning, where a heatmap may be created to highlight the capabilities in most need of improvement and investment. They are also used to map assets, such as IT systems, which enables better communication with teams involved in implementing strategies. Generally, however, business architecture and business capability models do not model specific processes as their aim is to be high level and agnostic to organizational structure.

Shortly after that, I was brought in to assist a client in the mortgage industry. As a mitigation strategy following the 2008 financial crisis, they had decided to outsource several critical business functions to a third-party provider. However, the project was in deep trouble, with status reports showing red on most measures despite a significant investment made in terms of dollars and resources. The requirements team lacked structure and was not making progress toward delivering a complete and high-quality product. Additionally, the business architecture team they had in place was struggling to add value. Recognizing the need for a strategic pause to recalibrate the approach, the senior client stakeholders gave us a three-week timeframe.

The business architecture team was implementing an approach I had not encountered before. They utilized a sophisticated modeling platform and connected their business capability model to detailed process models. During the pause, I collaborated with the leadership of the business architecture team to develop a comprehensive plan. This plan included a stakeholder engagement model, a rigorous approach to work product creation, developing standards, a quality assurance process, and even an estimating model for more precise delivery date projections. When the work resumed, we essentially created a Process Inventory, although that was not the name used at that time. We conducted interviews with functional areas to capture every process they performed and then created process models to detail how those processes were executed. These inventories and process models underwent a stringent quality assurance review to ensure compliance with standards before the identified process owner attested to their accuracy. This structure enabled the requirements team to measure their progress toward completion, gain a detailed understanding of the process operations, and obtain sign-off on final requirements from named process owners. By establishing this factory-like model with clear activity sequences and accountabilities, the project transformed from red to bright green. This experience taught me valuable lessons, particularly the anchoring power of this concept in large-scale transformation programs. I also recognized the importance of connecting business processes, maintaining high standards, and prioritizing quality as critical success factors.

From that point on, I sought to refine this methodology and test it in diverse scenarios, including an insurance spin-off, a bank merger, strategic architecture design for a wealth management firm, business continuity planning for a capital markets firm, and risk management for a global financial utility.

My focus shifted to selling this concept within the financial services industry, which put me in front of numerous clients with complex problems to solve. While many of these clients acknowledged the significance of process within their solutions, they struggled to articulate how it would play a role. For others, I seamlessly integrated it into proposals, knowing it would be beneficial. Regardless, I achieved success in selling the approach as its intuitive appeal resonated with everyone.

This chapter delves into the framework's details, encompassing various models and their interconnectedness, providing a solid foundation for transformation initiatives.

The Process Inventory Framework

In Chapter 1, I described how Process Inventory is a common language that facilitates strategic alignment, which is critical for transforming an organization into the digital age and operating with excellence. This is certainly true, but I'd like to go a bit deeper to describe what it represents.

The Process Inventory framework represents an ontology for an organization. The word “ontology” gained usage with ancient philosophers like Plato and Aristotle. “Ontology” comes from the Greek words “ontos,” meaning “being,” and “logos,” meaning “study” or “discourse” [3]. As they defined the term, ontology was the study of what exists and how the things that exist relate to each other, providing answers to fundamental questions about existence.

In modern times, ontologies are used in many fields such as information sciences, Semantic Web, bioinformatics, healthcare, natural language processing, artificial intelligence, and more. Today, the concept is defined as a formal representation of knowledge that encompasses a set of concepts, their properties, and the relationships between them within a specific domain. It provides a shared understanding of the structure and semantics of a domain, enabling effective knowledge organization, reasoning, and information integration [4]. To put this in layman’s terms for an organization, it maps out how all the resources and concepts intersect to support the mission of the organization.

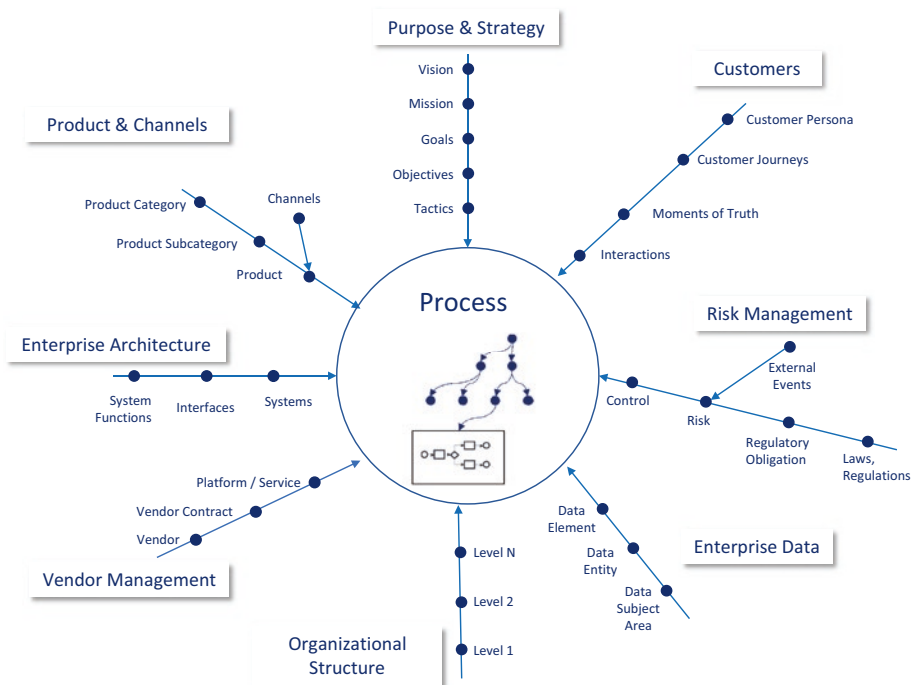


Figure 2-1. Organizational Ontology

Figure 2-1 is a visual representation of this ontology. As you can see, and this is not an exhaustive representation, there are numerous concepts that must come together to support an organization. Organizing knowledge in a specific domain is typically done through a taxonomy. Taxonomies are classification systems in which entities in domains are placed in hierarchical categories that are exclusive and exhaustive. Exhaustive means that every possible category, especially at the top of the hierarchy, is represented, and exclusive means that every entity belongs to only one category. To illustrate this, I'll use the classic example of the animal kingdom taxonomy. At the top level of the hierarchy, each animal is classified as vertebrate or invertebrate. A level below vertebrates are reptiles, birds, fish, mammals, and amphibians. The levels go deeper, but this taxonomy is exhaustive as there are no other high-level categories. It's exclusive in that any animal can exist in one and only one category.

As represented in Figure 2-1, each of the concepts that support an organization is its own taxonomy, which makes the ontology for an organization a polycategorical ontology, meaning knowledge is to be understood in how multiple taxonomies intersect. In polycategorical ontologies, entities are distinguished by how fundamental they are to the ontology, meaning how dependent their existence is relative to other entities. I discussed in Chapter 1 that the organization's purpose and strategy are the most fundamental to the existence of everything, but after that, the process is fundamental to all capabilities of the organization. For instance, an accounting system wouldn't be needed without the need for accounting processes.

The earlier ontology definition includes understanding the relationships between concepts within a specific domain; in this case, the domain is the organization. So, a consistent approach for mapping the concepts between these taxonomies is needed to have a coherent representation of knowledge. Since process is the most fundamental to the capabilities of the organization, it must be the top-level concept, meaning the entities in other taxonomies need to be mapped to how they support processes in the Process Inventory taxonomy. That is how process becomes the common language of the organization within this ontology.

Since organizational units are constructed to support the need of processes, this ontology explicitly maps disparate concepts to the Process Inventory taxonomy, providing horizontal alignment of knowledge and information.

For the remainder of this book, I will refer to the integration of these concepts as **process metadata**, which is information about the process. There are two broad categories of metadata. The first is operational information metadata, which provides information on the process's existence. This includes the products it supports, the legal entity it belongs to, or who owns the process, or the business capabilities it represents. The second is resource metadata, which details the resources leveraged in the execution of the

process. This includes systems, data, channels, organizational units, third-party vendors, etc. The implication for modeling is that operational information metadata can be designated at the taxonomy or process definition level, while resource metadata can be further detailed through process models.

Figure 2-2 shows the same model but framed hierarchically. Since Process Inventory is constructed following the organizational structure, then it provides views that are relevant from the CEO through a low-level individual contributor, which is key to vertical alignment.

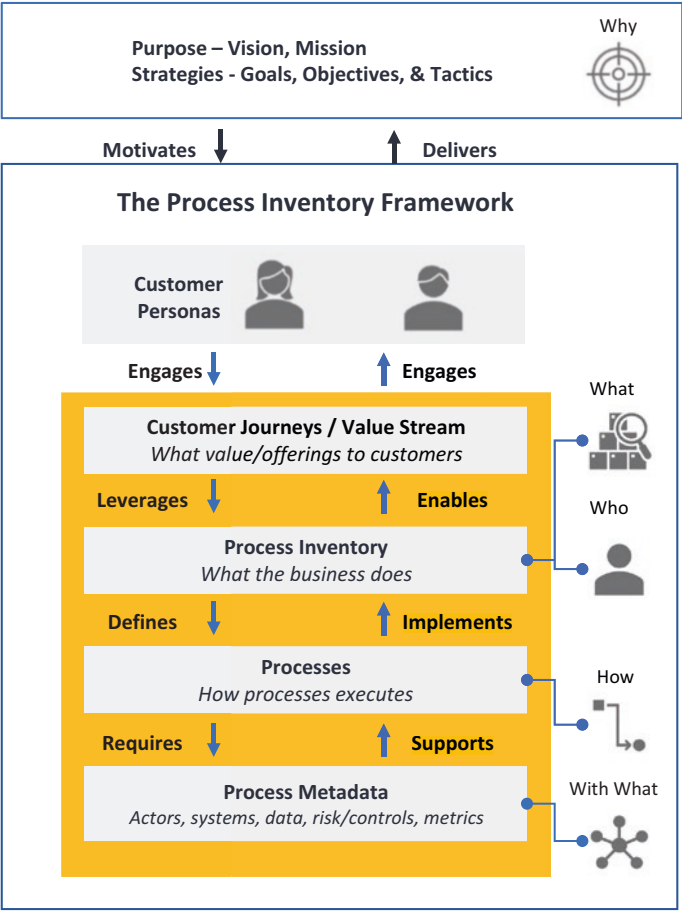


Figure 2-2. The Process Inventory Framework

This framework clearly delineates aspects of the 5W (and I H) model, which are crucial for understanding the motivational system of the organization. This provides a tool that helps organizations connect the “Why” in terms of

their purpose and strategies to the “What” processes they execute to achieve their goals, the “Who” responsible for the results, and the “How” those processes are performed, as well as “With What” resources are leveraged.

The framing in Figure 2-2 will be leveraged in the rest of this chapter to describe various parts of this framework.

Process Inventory

Definition

Merriam-Webster defines a process as “a series of actions or operations conducing to an end.” I point out this definition to highlight that this framework takes a generous definition, meaning that the goal of Process Inventory is to identify every action that is performed by anyone in the organization, any customer-facing technology systems, and any outsourced third-party providers.

The result of capturing all processes is an **end-to-end process framework**. I define this through the characteristics of (1) capturing everything an organization does and (2) being able to model processes from start to finish as they traverse different parts of the organization.

Here are some characteristics of Process Inventory:

- **Comprehensive Coverage:** The goal of Process Inventory is to capture everything that a business does within an identified scope. This is achieved through an interview process that aligns with the organizational hierarchy as highlighted in Figure 2-3.
- **Accuracy Through Attestation:** Since Process Inventory is constructed through the organizational hierarchy, this provides ownership to review the accuracy of the model after its initial creation and in each subsequent maintenance review. Asking stakeholders for formal attestation encourages them to pay attention to the relevant details and ensure accuracy.
- **Alignment with Strategy:** The transparency of processes enables a clear line of sight in how each process contributes to the achievement of the organization’s goals. It can be assessed in terms of its strategic importance and performance metrics.

- **Top-Down Decomposition:** High-level activities as seen by the CEO or senior leadership are made up of subprocesses, and those subprocesses can be made up of other subprocesses. This model provides the ability to drill down to the appropriate level of detail given the encountered problem.
- **Models Current State and Future State:** Capturing Process Inventory is typically done on the current inventory of processes, but it can be useful to show future state processes when executing a transformation program.
- **Process Levels:** Since this inventory is constructed by leveraging an organizational hierarchy, there is no set level for documenting process models, such as Level 3, as different organizational hierarchies can be built at different levels of depth.

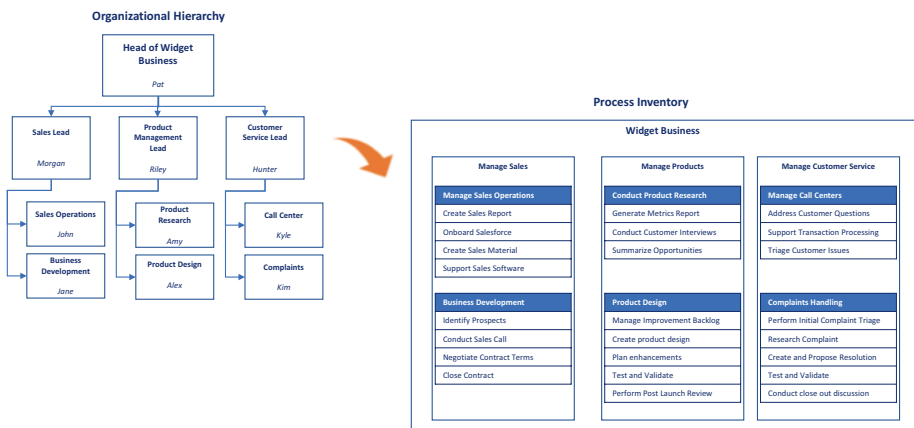


Figure 2-3. The Process Inventory Identification Process

Quality Is Paramount

Since Process Inventory serves as the common language for organizing organizational knowledge, ensuring quality is crucial. These models contribute to a repository of information interconnected with other process metadata. When stakeholders begin utilizing this information in their roles, encountering significant quality errors can undermine their confidence in the effort. Overcoming negative perceptions of poor quality can be challenging.

For each model, I will outline an approach to construction that includes multiple quality review points and stakeholder sign-offs. Adopting these quality checkpoints is of utmost importance to prevent costly or fatal mistakes. It is considered a best practice to conduct internal standards alignment reviews before seeking stakeholder attestation, thereby avoiding the need for them to re-attest if quality issues are discovered.

Process Naming

One principle that is important when constructing the inventory of processes is to provide clarity regarding the actions taking place. A good test for that is whether someone not familiar with the business unit or functional area can intuitively understand what the process accomplishes just by reading its name. One effective way to achieve this is by adopting a verb + noun naming convention. Since processes represent actions, the verb specifies the action, and the noun represents the entity on which the action is performed. Examples of good process names include “Open Account,” “Submit Payment,” and “Document Dispute Resolution.” Additionally, when naming processes, it is advisable to minimize the use of generic verbs, such as “Manage,” as they lack precision in describing the action being taken.

Furthermore, it is crucial to use the language of the business when engaging stakeholders as they are being asked to take ownership and be accountable for performance. The process names should align with their existing vernacular. For instance, if they refer to a process as “Onboard Client,” it is important not to force them to call it “Open Account” just to match the terminology used by other business units.

In my experience, process owners value consistency and readability that standards bring while still allowing flexibility to align with their preferred vernacular.

Maintenance

When embarking on this work, one of the initial concerns raised is how to maintain the model’s accuracy. This is a significant question, considering the constant changes that occur within many organizations, including the deployment of large-scale programs and organizational restructuring. Addressing this concern involves two key components.

Firstly, each process area must undergo periodic attestation, such as yearly, wherein the process owner identifies and highlights any changes and subsequently attests to the revised model. This ensures that updates are captured and acknowledged. Periodic attestations are not a new concept as they are used in several topics such as access controls, financial reports, and compliance reporting just to name a few.

The second aspect involves integrating this model into the organization's business management practices. Part 2 of this book will delve into this topic, encompassing the management of efficient operations, change management, risk management, and technology environment management. By instilling a sense of shared ownership among stakeholders, with everyone actively invested in maintaining an up-to-date model, integrating and attesting changes on a near real-time basis becomes more streamlined.

Process Ownership

A significant advantage of this framework lies not only in identifying the business processes within an organization but also in providing details regarding accountability for each process and process area. Process owners hold responsibility for executing processes and, in many cases, for identifying and driving changes to those processes. Since Process Inventory is built upon the organizational hierarchy, none of these accountabilities are new. However, adopting this framework enhances transparency by clarifying ownership and may lead to more defined performance metrics for both processes and their respective owners.

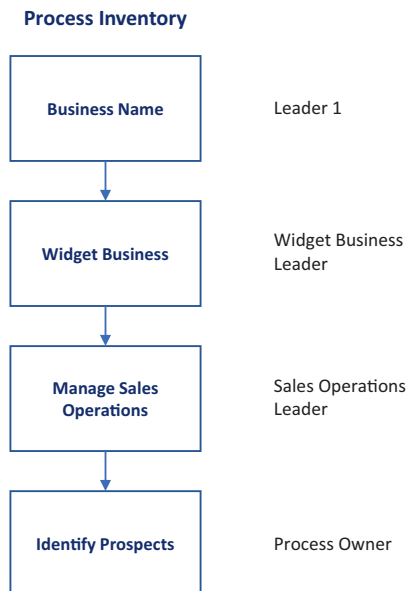


Figure 2-4. Process Ownership Model

As depicted in Figure 2-4, every point in the Process Inventory taxonomy will have an identified owner. This implies that the entire reporting structure will bear some level of accountability for process performance.

This structure facilitates easier maintenance of the Process Inventory when ownership transitions occur. In such cases, contacting the leader higher up in the chain allows for the identification of new owners.

Furthermore, this structure not only brings transparency to process accountability but also enables the cascading of goals throughout the organization. It ensures that lower-level teams have a clear understanding of how their efforts contribute to the organization's overarching objectives.

Omnichannel Processes

An exception to identifying processes by following the organizational hierarchy is with omnichannel processes. Many products that organizations offer allow for multiple customer contact points via various channels. These channels can include in-person interactions through physical stores or phone calls, as well as digital interactions through websites, mobile phones, or other electronic interfaces. Omnichannel aims to provide a seamless and effortless, high-quality customer experience that occurs within and between contact channels [7]. This can be achieved by providing a consistent experience to customers across all channels. However, it can become much more sophisticated by offering customers a hyper-personalized experience that leverages insights gained from the knowledge of the customer, including tracking their behavior as they interact across channels. This may also involve providing customers with the ability to start a transaction in one channel, such as the Web, and seamlessly continue it in another channel, such as physically going to a store.

A critical success factor in implementing the organization's omnichannel strategy is having a clear inventory of interactions, and by which channel, customers can interact with for a product.

A starting point for this is the product manager for each product. They can provide the list of transactions or processes associated with that product and indicate the channels in which those processes can be executed. This list should then be included in the Process Inventory for each channel. Having consistent process names across channels will enable teams to harmonize behaviors and collaborate with technology experts to develop the necessary back-end platforms that facilitate the omnichannel strategy.

Outsourced Processes

Many organizations outsource business processes to vendors for various reasons, such as not considering them part of their core value-add or lacking the necessary skills and resources. It is crucial to note that while the execution of certain processes may be delegated to a third-party vendor, the accountability

for their performance remains with the organization. This means that if a vendor experiences a high-profile data breach or performance issue that impacts operations or attracts regulatory attention, the originating organization is held responsible. To address this, it is important to include the outsourced vendor processes in the Process Inventory.

Depending on the contractual arrangement, the vendor may not provide specific details on how the processes are executed. However, having clarity on the list of outsourced processes enables better monitoring of service level agreements (SLAs) and facilitates stronger management of these vendors. By monitoring performance against agreed-upon SLAs, organizations can proactively address any issues, ensure compliance, and maintain effective oversight of their outsourced processes.

Incorporating these considerations into the Process Inventory helps organizations maintain transparency, mitigate risks, and uphold their accountability even when processes are outsourced to external vendors.

Integration with Business Capability Model and Standard Process Models

In the definition I provided on business architecture and business capability models earlier, I mentioned that they are largely independent of specific organizational structures. There are also standard process taxonomy models for various industries, such as the APQC Process Classification Framework, which have adopted the same principle of being agnostic of specific organizational structures. This means that a capability or a process will be represented once and only once in those models regardless of how many business units or functional areas execute those processes.

However, this poses a challenge when trying to rationalize overlapping processes, standardize how certain processes are executed, or identify risks and implement effective controls. Leadership and practitioner teams can lose sight of these idiosyncrasies.

That doesn't mean these models lack value. They are extremely valuable in highlighting similar processes across disparate business units or functional areas. Since the Process Inventory leverages the specific vernacular of the business, it may not be immediately apparent that processes across different groups are performing the same tasks.

The ideal solution is to leverage both models and map them together, providing each stakeholder with the information they are seeking. For example, a risk team that needs to understand all the processes executed by an organizational unit will see all the processes in that area and how their execution may pose risks. On the other hand, a design team aiming to rationalize processes to

create a shared services organization will gain insights into the process owners, their unique requirements, and how those must be implemented in the new organization.

Table 2-1. Highlights the Main Differences Between Process Inventory and a Business Capability or Standard Process Model

Process Inventory	Business Capability or Standard Process Model
• Constructed via the organizational hierarchy • Represents a complete inventory of all the processes that a business unit or functional area performs • Changes with organizational or process changes • Requires interviewing the business	• Agnostic of organization structure • Represents a complete inventory of all that a business can do • Stable and only changes with significant organizational changes • Can be obtained and leveraged immediately

Figure 2-5 shows the APQC Process Classification Framework. This diagram is their industry-agnostic model, but they also have tailored ones for different industries. The standardization is clear from the top-level view. This model is agnostic to organizational structures.

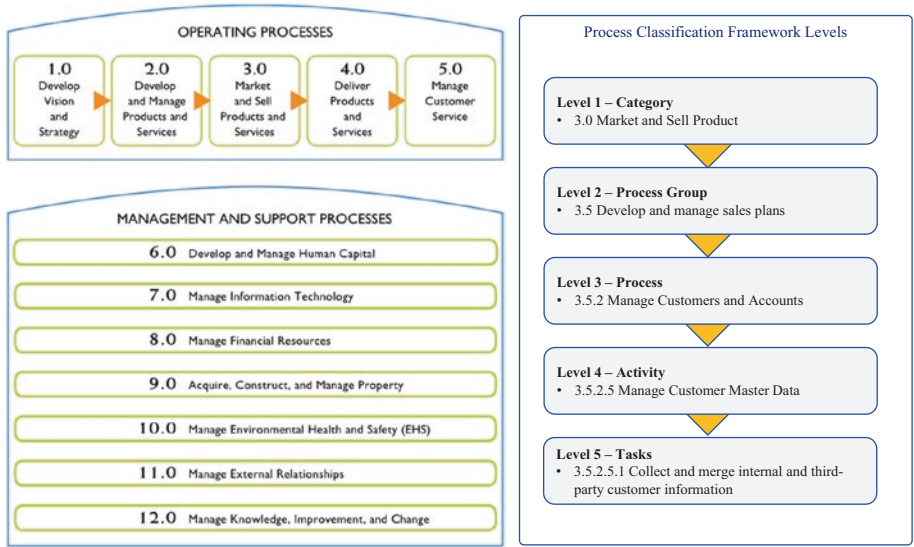


Figure 2-5. APQC Process Classification Framework

Figure 2-6 illustrates a real-world situation which is relatively simplistic, in which two different business units of the organization have sales teams. In many cases, especially with organizations with multiple business units or geographies, it can be hard to understand the overlap. Also, as you get deeper within an organization, it's possible that you'll find processes that are performed in an organization that are outside of the original mission of that group. An example is when a business unit creates a shadow IT [5] organization because they perceive or experience limitations in the central IT organization. Deciphering the details and performing alignment across the enterprise is made possible when both models are leveraged together.

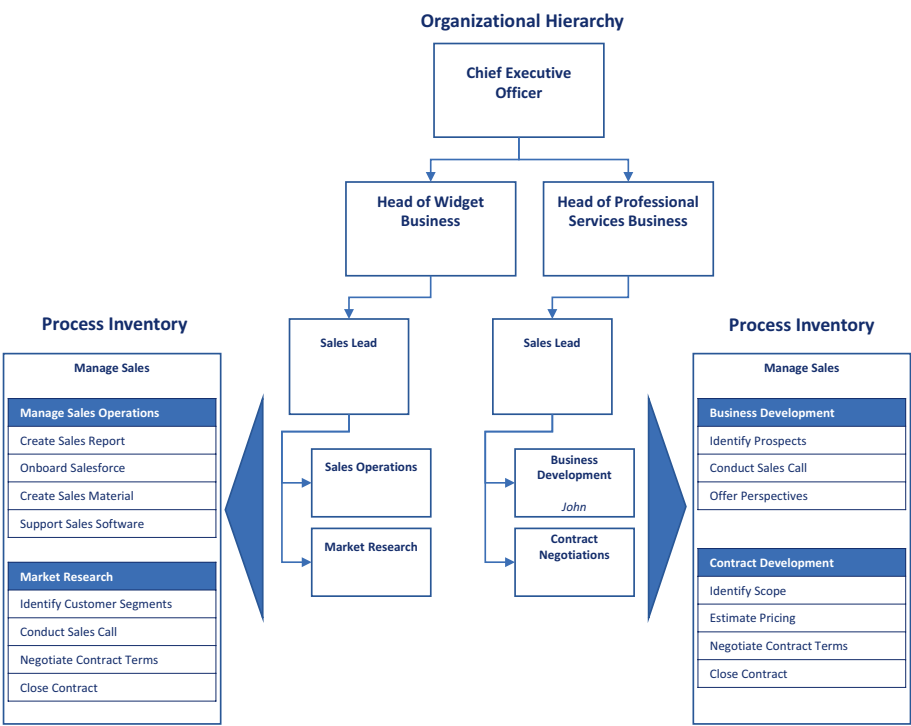


Figure 2-6. The Process Inventory Identification Process

The solution is illustrated in Figure 2-7. The Process Inventory is created then each process identified is mapped to the lowest level available and appropriate in the Standard Model.

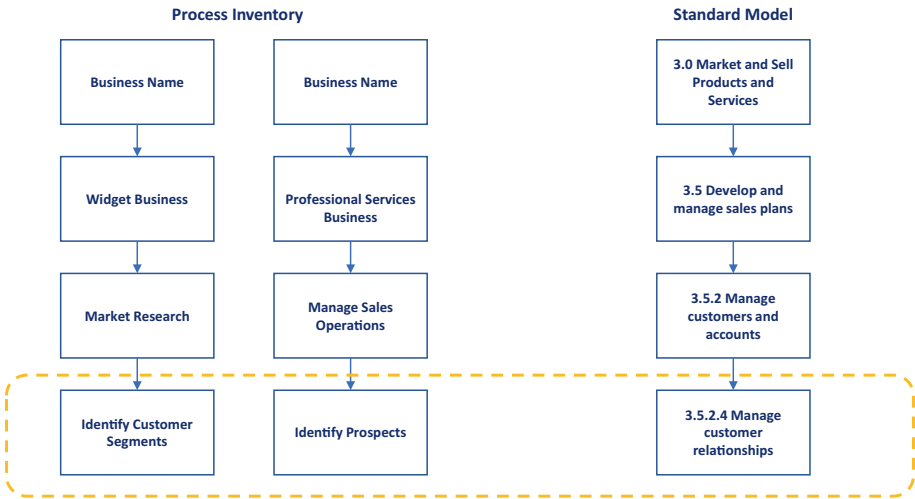


Figure 2-7. Linking Process Inventory with Standard Model

This best-of-both-worlds approach has many benefits across many use cases. It enables reporting through either lens. This is particularly useful for organizations that have become complex and are looking to optimize their operating model through optimized processes and reduced costs of doing business.

Process Mining

Process mining is an emerging concept that enables the understanding, analysis, and optimization of processes using data collected from system event logs. The first commercially available process mining tool became available in 2011, and there are now over 35 vendor tools [6] offering this capability. One of the key features of process mining tools is process discovery, where the tool automatically creates a graphical process model based on the information extracted from event logs.

While these tools offer valuable features, they should be considered as a complementary addition to manually collected Process Inventory. Process mining can be complex to implement, requiring resource-intensive effort and time, especially if an organization has a large number of systems with diverse log file formats. Implementation may necessitate changes to existing systems, data collection mechanisms, and data quality improvement efforts.

These tools do not aim to create a comprehensive and exhaustive set of all processes performed by an organization. Even if they could, organizations would still need to verify and attest to the accuracy of the output.

To fully benefit from process mining tools, the output should be harmonized with the existing Process Inventory, allowing for the alignment of metadata. This integration facilitates a more comprehensive understanding of the processes.

Conformance checking is another important feature provided by process mining tools. It involves identifying discrepancies between what humans document in the Process Inventory, process models, or procedure documents and what occurs in practice. These insights can be used to address gaps in compliance and improve the quality of documentation.

Additionally, process mining tools can analyze process performance data, including metrics such as throughput time and bottlenecks. This information serves as input for teams involved in optimizing these processes.

Process Inventory Metadata

When working with process owners to create the Process Inventory, it's important to gather as much descriptive information about the processes as possible. This information should be included in the packet when the process owner provides attestation. Table 2-2 outlines the types of operational information metadata to collect during the creation of the Process Inventory. Resource metadata, such as systems and people, can also be collected at this stage, but I will discuss that further when addressing process modeling and metadata.

Table 2-2. Highlights Metadata That Should Be Collected When Creating a Process Inventory

Metadata	Description	Source
Process Description	This should be a succinct description of the process which is understandable by everyone.	Process Owner
Process Owner	The name of the owner of that process.	Organizational Chart
Products	The product or products that the process supports.	Product Catalog
Channels	The channels or channels that the process supports.	Channel Inventory
Standard Model Alignment	The business capabilities or standard processes which each process aligns to.	Business Capability Model or Standard Process Model
Risk and Controls	The risks and control that align to the process.	GRC Risk Data Repository

Sourcing Process Metadata

To deliver on the promise of this framework, which aims to structure all knowledge and information for the organization in an ontology, data from numerous authoritative data sources need to be brought together and aligned for processing. An authoritative data source is a trusted and reliable repository of information that is widely recognized as the official or primary reference for a particular subject or domain.

In this section, I will detail how this data is conceptually linked and how the movement of data across sources facilitates this alignment.

Integrated Process Metadata Model

Figure 2-8 illustrates a conceptual model for bringing this together. The Process Inventory identifies every process that occurs in the organization in a hierarchical manner down to the process name. Operational information metadata, which provides information about the existence of each process, can be aligned with the identified processes.

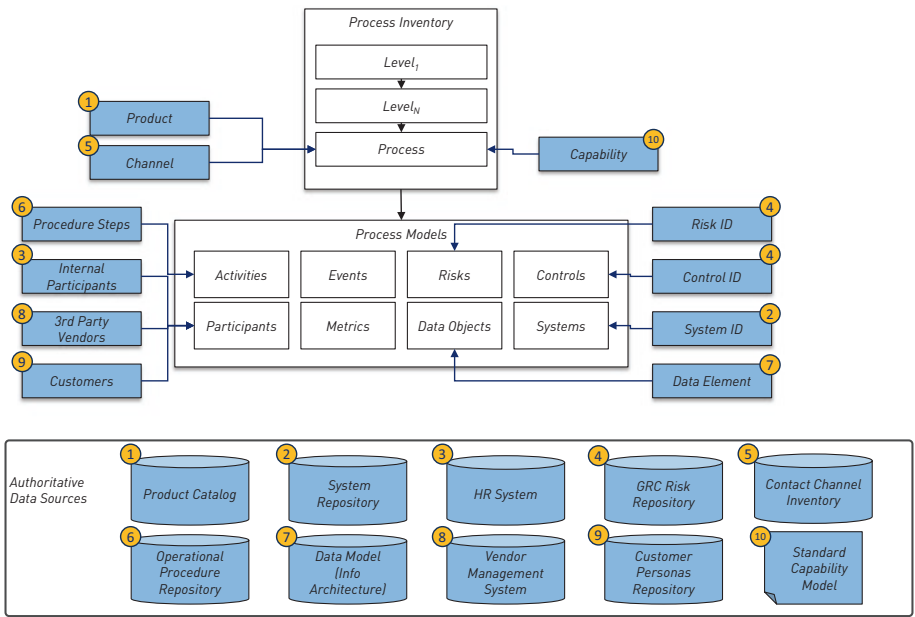


Figure 2-8. Linking Process Inventory with Standard Model

Process models are created to graphically represent how the processes are executed. Figure 2-8 shows the typical components in a process model, such as activities, participants, and systems. Many of these components represent

the resources leveraged in conducting the process. This resource data is typically managed by different parts of the organization. For instance, systems are managed by IT, and risk data is managed by the risk organization. However, the ability to connect these concepts across sources has largely been absent.

These data sources typically contain a rich set of metadata about their resources. Connecting the identifiers of this data enables valuable insights. For example, connecting an activity in a process with the system that the activity leverages enables the organization to understand the programming language and the data centers that support that activity. This is very powerful for tasks like identifying the root cause of an issue, detailing the impacts of change, or determining risks that need to be mitigated. This is how the ontology is constructed for the organization, with the Process Inventory being the fundamental taxonomy.

Conceptual Architecture

Figure 2-9 represents the conceptual architecture to support the linking of data within the Process Inventory framework.

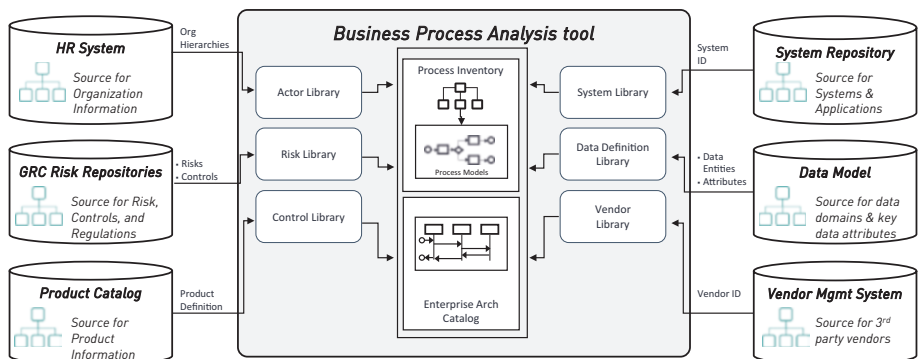


Figure 2-9. Conceptual Architecture

A **Business Process Analysis** tool is required to build and manage the Process Inventory and the library of process models associated with it. The features and a list of vendor tools containing this functionality will be detailed in Chapter 10.

Data, along with descriptive information, is migrated from authoritative data sources into the tool to construct libraries. Initially, there is a load of data, but periodic refreshes are necessary to prevent the libraries from becoming outdated. When the Process COE team creates the Process Inventory model and its associated process models, they connect the elements in these libraries to the appropriate locations within the framework. This connection of

information to processes establishes the Process Inventory as the common language that links various aspects of how the organization operates. In this book, I will refer to this tool and the information it contains as the **Integrated Operational Repository** since it becomes the authoritative data source for aspects of the organization’s operations.

It’s important to note that the quality of the data sources is critical. If the data is untrustworthy and of poor quality, it is strongly recommended to postpone importing that data into the repository until those issues can be addressed. Poor data quality would undermine the intended purpose and could create negative perceptions of the repository.

Exporting the Common Language

Now that we have established the Process Inventory as the organization’s common language and connected operational information through metadata, it is logical to export this common language to other systems that generate data. This will enable the seamless association of data across all internal systems.

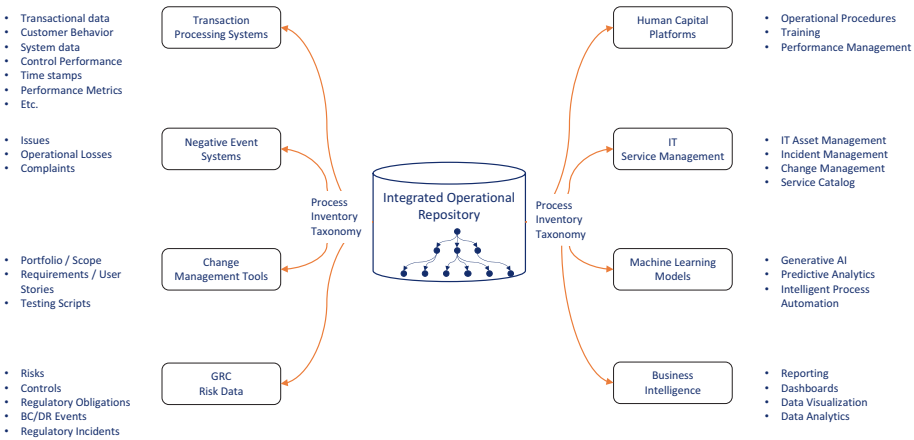


Figure 2-10. Publishing Process Inventory Taxonomy

Figure 2-10 is not an exhaustive list of systems that can align their data with the Process Inventory, but it should give you a sense of the benefits, which include

- Aligning negative events such as issues, incidents, and operational losses to process data gives triage teams access to underlying resources, reducing triage time.

- Aligning risk data results in higher-quality risk assessments, with named process owners accepting accountability for control performance. This enables more accurate reporting on the risk landscape for the board and senior leadership.
- Aligning change management deliverables brings clarity of scope, asset reuse, and stronger budget and timeline management.
- Aligning transactions in transactional processing systems provides the ability to create near-real-time monitoring dashboards, highlighting processes that are not meeting expectations.
- Aligning with human capital systems enables clarity of procedures, deployment of training, and alignment of performance metrics with process goals.
- Aligning this data with machine learning models allows for the identification of patterns that may not be accessible to humans, thereby improving numerous operational metrics.

Process Modeling and Metadata

The industry-accepted standard for documenting business processes is Business Process Model and Notation (BPMN) 2.0. BPMN 2.0 is a standard maintained by the Object Management Group (OMG). The primary goal of BPMN is to provide a notation that is easily understandable by all business users, including business analysts creating initial process drafts, technical developers implementing the technology for those processes, and businesspeople managing and monitoring the processes [8].

Most, if not all, Business Process Analysis tools on the market support native documentation of processes in this standard.

The complete BPMN 2.0 specification can be found on [omg.org](https://www.omg.org), and there are numerous online training resources available. My intention is not to provide education on the standard but rather to highlight how the standard enables the alignment of metadata and the implications of aligning this metadata for various use cases.

Figure 2-11 is a sample process model highlighting key components of a process model. In this section, I will highlight how the structure of the BPMN 2.0 process model is a great way to integrate this metadata in its proper context.

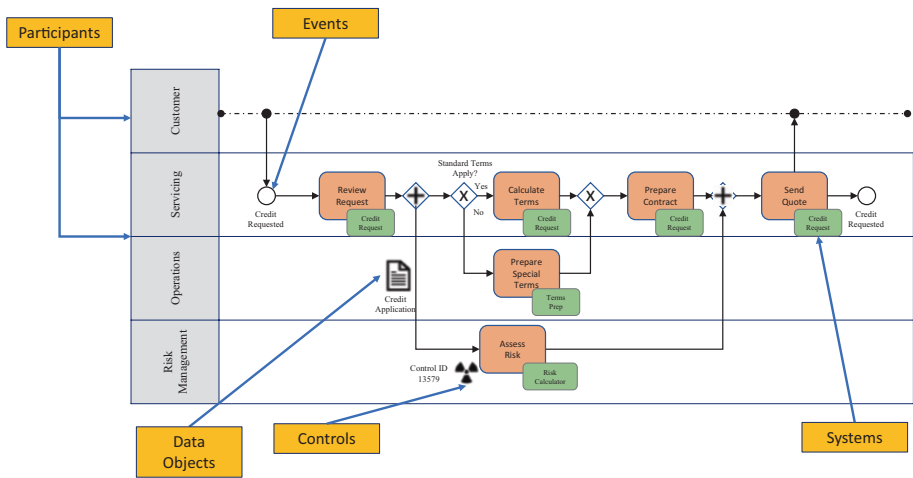


Figure 2-11. Sample BPMN 2.0 Process Model

Leveraging Metadata Libraries

When modeling processes in the modeling tool, it's important that data is standardized across models. This consistency will enable the ability to perform impact analysis across processes that leverage certain resources. To facilitate this, only data aligned in the libraries, as depicted in Figure 2-9, should be leveraged. This means hand-creating net new objects needs to be discouraged if there is an established library to choose the appropriate item.

Participants

Process models in BPMN 2.0 include lanes, which represent the participants in the process. Lanes can exist within pools, which represent the organizational units that contain the participants.

There are two types of lanes in BPMN 2.0:

- **White Box Lanes:** These lanes are used to explicitly document the activities and tasks of participants internal to the organization. They provide transparency and detail the handoffs between departments.
- **Black Box Lanes:** These lanes are used to show interaction points with external participants, typically customers or external vendors. In these cases, it may not be necessary or possible to identify their specific activities, and the focus is on the interaction points.

Both white box and black box lanes serve specific purposes based on the need for transparency and level of detail in representing the activities and interactions within a process.

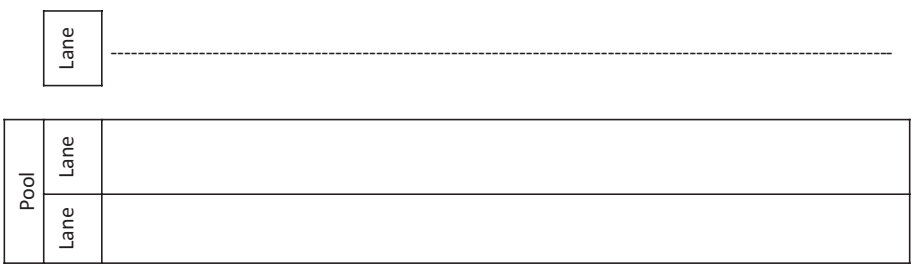


Figure 2-12. Participant Lanes

There are three types of participants that are modeled through these lanes: customers, third-party vendors, and internal teams. Here are the details of how each is sourced:

- **Internal teams** are sourced from organizational charts within the HR systems. Process modeling teams will have to determine the appropriate level of granularity to use. If lanes are defined at too low of a level, then there may be numerous lanes which will make the process models too hard to read.
- **Customers** are typically sourced from the customer experience team if one exists. Customers typically align to products or services being delivered by the organization, and they can be further segmented by the profile of the customer persona.
- **Third parties** are sourced from a vendor management system which is typically maintained by the procurement or the risk team. The focus here is on third parties that perform processing on behalf of the organization.

Figure 2-13 illustrates the load of data from their respective authoritative sources of data into the Participant Library for process modelers to use when creating lanes in process models.

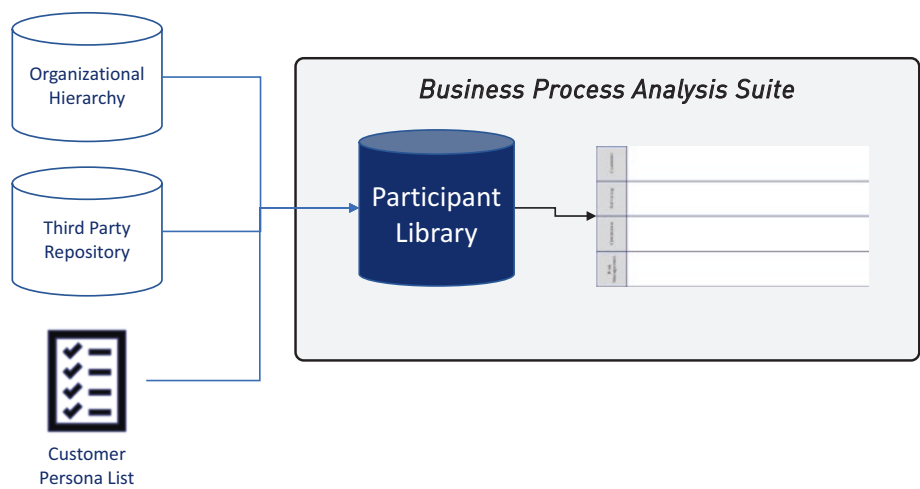


Figure 2-13. Participant Data Load

Events

Events represent occurrences that occur during business process execution. Unlike other metadata elements covered in this section, events are not sourced from an external authoritative repository; instead, they are identified through interviews with process owners and documented in process models. By building process models derived from a comprehensive list of processes provided by the Process Inventory, a powerful repository can be created for various purposes, including

- **Event-Driven Architectures:** A software design pattern in which integration and system actions are triggered through events.
- **Data State Changes:** Exclusive changes in data states occur through events. For example, a loan can transition from the application state to the approved state through an event named “Loan Approved,” which alters the underlying data state in the appropriate data stores.
- **Operational Reports:** Leveraging events, operational reports can be created to provide insights into the volume and performance of underlying processes.

Events are also critical in facilitating **end-to-end process modeling** as an end event in one process can trigger the start event in another process, potentially in a different area of the business.

The events represented in business process models are only those that are visible to process owners. There are also technical events, such as data being transformed and migrated through a batch process, which will be covered in Chapter 6. When combined, these events create a powerful repository that can drive valuable insights and designs.

For this event repository to be useful, it is important to have descriptive event names. The recommended naming standard is a noun plus past tense verb, such as “Loan Approved,” “Contract Signed,” and “Report Delivered”.

Systems

Most activities in today’s digital age occur on systems. Mapping activities that occur in a business process to the system that supports those activities supports transparency of IT and facilitates better dialogue between the business and technology teams.

This level of transparency enables a rationalized Enterprise Architecture design, which will reduce complexity and redundancy. It will lead to a lower cost for maintaining the IT environment and enable greater speed to market for change items.

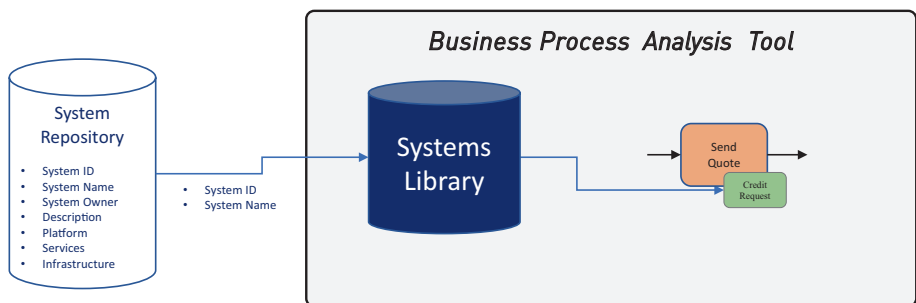


Figure 2-14. Systems Data Load

Figure 2-14 highlights how information is loaded from the source repository to the Systems Library. Each activity that leverages a system is tagged with the specific system from that library.

Data Objects

Managing the vast amounts of data that organizations generate and are required to maintain has been an ongoing challenge for most organizations. As a result, many have established dedicated data governance teams to establish

and enforce policies, procedures, and guidelines for the management, quality, and security of their data assets.

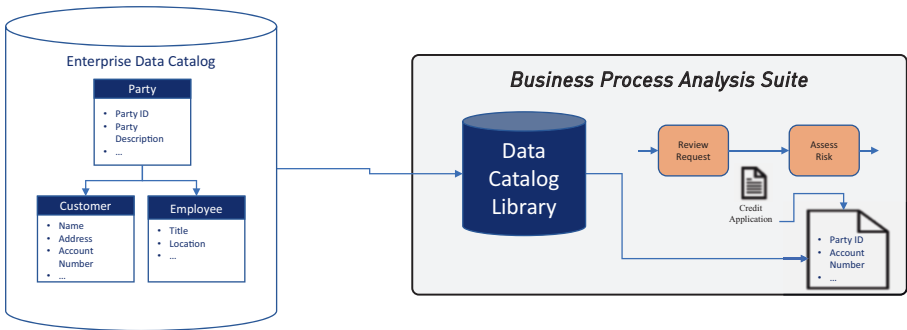


Figure 2-15. Data Catalog Data Load

Figure 2-15 illustrates the loading of data definitions from the Enterprise Data Catalog. One of the key goals of data governance is to align terms and definitions across the organization, enabling consistent application of policies and effective data stewardship. Enterprise Data Catalogs play a crucial role in achieving this consistency by identifying and cataloging the data elements used in business operations.

In process modeling, data is utilized during activities or handoffs, and this data is represented and captured through data objects. The naming convention for data objects is to align them with the business’s terminology for the corresponding data.

Business Process Analysis tools enable modelers to identify the data elements associated with each data object. By aligning these data elements with the data identified in the Enterprise Data Catalog, consistency and collaboration can be promoted across different groups. For example, if one group refers to a data element as “Social Security Number” while another group calls it “SSN,” mapping these terms ensures they are both referring to the same item.

This level of clarity regarding which data elements are associated with specific processes provides data governance teams with transparency when applying their policies, procedures, and guidelines.

Controls

Risk is an inherent part of conducting business. The critical aspect for any organization is to have a precise understanding of the risks involved so that leadership can make informed decisions on which risks to accept and implement appropriate mitigation strategies. Risks can arise from various sources, including operational risk, compliance risk, business continuity, and others, which may be triggered by internal or external events.

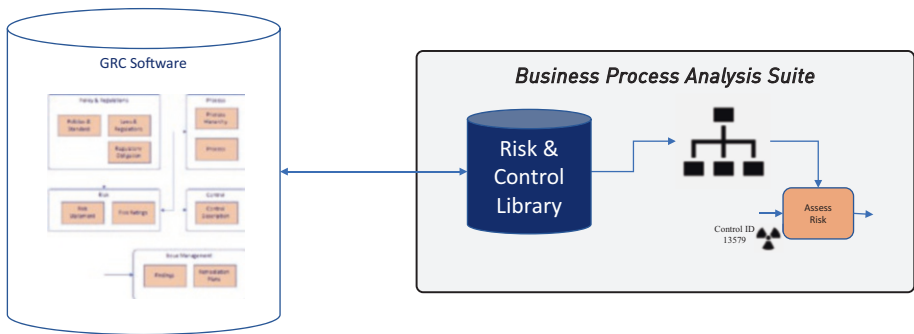


Figure 2-16. Data Catalog Data Load

Figure 2-16 illustrates the loading of risk and control information from the risk repository. Effective risk management requires a thorough comprehension of the relationship between the organization's processes, the risks associated with those processes, and the controls implemented to mitigate those risks. It is essential for organizations to maintain their risk data to perform risk assessments, design and monitor controls, and generate risk reports for the board and senior leadership. Typically, this risk data is managed using a GRC (governance, risk, and compliance) tool.

Explicitly linking risks to the processes that generate or are impacted by them brings several benefits, such as enhancing the completeness of risk assessments, aligning process owners' accountability for control performance, and improving monitoring and reporting capabilities.

This process-risk linkage also creates a two-way flow of data. The GRC tool benefits from having a comprehensive inventory of processes as part of its data model to ensure completeness, while the modeling team, in collaboration with process owners, gains insight into the specific controls they need to design and integrate into their processes.

Customer Personas and Customer Journeys

Providing customers with a great experience is key to driving customer loyalty, differentiating from competitors, improving brand reputation, and gaining valuable customer insights that lead to increasing revenues. This drives many organizations to anchor their digital transformation program on the goal of improving customer experience.

This is why many are creating dedicated customer experience (CX) teams. These teams have adopted an ethnographic research methodology that provides them with a deeper understanding of customers' needs, expectations, and behaviors. This outside-in analysis gives organizations valuable insights into how they need to change to meet their customer experience aspirations.

These CX efforts typically focus on understanding customers through customer personas and documenting their experiences interacting with products and services in a customer journey.

Customer personas are fictional representations of customers who use your products or services. They often include information about demographics, lifestyle, behaviors, goals, motivations, pain points, and communication preferences.

Customer journeys are the end-to-end set of interactions that a customer has with a product or service. They usually include information about the experience that ties to the customer persona, such as touchpoints, customers' needs, emotional reactions, and pain points.

Understanding customers through these artifacts is imperative in defining the CX strategy. Organizations can choose to reinvent the customer experience entirely or focus on tactically addressing current pain points. Regardless, they provide a basis for understanding customers and the organization, which should lead to success.

Astonishingly, only 25% of CX professionals say their companies' CX programs improve customer experience according to a study by Forrester [9]. There are many reasons for this, but one of the most prominent is the failure to identify the root causes of poor CX.

Even if the root causes can be understood, coordinating change across the organizational silos that need to collaborate to implement the change is a significant undertaking.

Linkage to Process Inventory Framework

A great characteristic of customer journeys is that they capture every touchpoint a customer has with an organization throughout their relationship life cycle. This is insightful as it covers the end-to-end process for the product or service, crossing organizational boundaries such as marketing, sales, back-end operations, channels, and technology platforms.

Since the Process Inventory aims to create a complete and accurate inventory of the organization's processes, it is important that each touchpoint identified in a customer journey is represented in the Process Inventory. This relationship is illustrated in Figure 2-17. By mapping the customer journey interactions with the processes in the Process Inventory and aligning metadata, it enables traceability from the customer's perspective through each resource that supports those interactions.

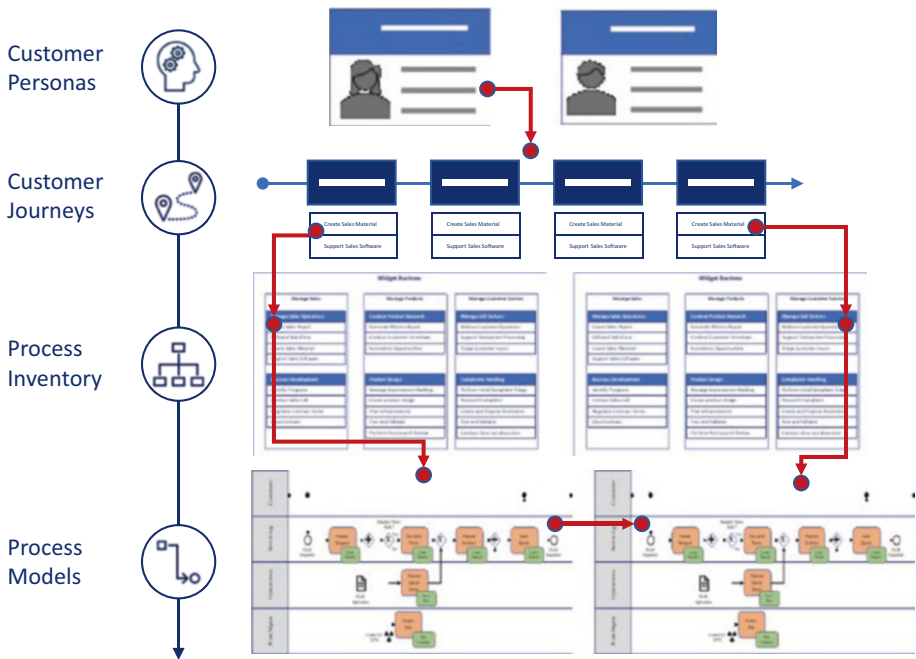


Figure 2-17. Customer Journey—Process Inventory Alignment

This approach allows for precise identification of root causes, enabling the development of appropriate solutions. However, implementing such changes may not be trivial as it may require coordinating change efforts across multiple lines of business, geographies, and practitioner teams.

The changes, along with their comprehensive set of impacts, can be estimated and incorporated into the change portfolio investment process. This provides the change investment committee with a solid business case for prioritizing and allocating discretionary change expenditure related to customer experience. If funded, this ensures that all practitioner teams are aware of the full range of impacts and have clear accountability for implementing the required changes.

Many organizations strive to build a culture of customer centricity, but sometimes this goal may be lost on teams that are several levels removed from direct customer interactions. In such cases, it becomes challenging for them to establish a direct connection between their day-to-day tasks and their impact on the customer. The structure described earlier provides the necessary clarity, enabling everyone to be accountable for their part in delivering an enhanced customer experience.

Value Streams

Value streams are the end-to-end process from initial conception through to delivering value to an end customer or stakeholder.

Value streams and customer journeys are very similar in that they are both end-to-end views that cross organizational boundaries and leverage specific processes identified in the Process Inventory. The main difference is that customer journeys are from a customer's perspective, an outside-in view, while value streams are from an internal perspective, an inside-out view. The modeling standards and techniques for connecting to specific processes, as depicted in Figure 2-17, should be consistent.

Key Takeaways

- **Ontology** is a construct for representing an organization's knowledge. The Process Inventory serves as the anchor framework, ensuring consistent representation and alignment to drive business value from this knowledge.
- **Process Inventory** is a framework designed for capturing and managing a comprehensive and accurate inventory of the processes performed within an organization.
- **Metadata** is sourced and aligned from various authoritative data sources to provide context for operational information and the utilization of resources within the processes they support.
- **Additional models** and modeling techniques are required to understand the organization from diverse perspectives. These include BPMN process models, customer personas, customer journeys, and value streams.

In the next chapter, we will explore key trends in digital transformations as well as understand the components of a successful digital transformation program.

Defining the Digital Transformation Program

We'll explore the critical role of building a digital strategy as the foundation for successful digital programs. When articulating a digital strategy, it's essential to consider key trends that are reshaping business models and transforming how organizations operate. These trends encompass new innovation models, a heightened focus on the customer, and the recognition of data as a strategic asset.

We will delve deeply into a methodical approach for establishing the digital transformation program and crafting a transformation roadmap aligned with an organization's unique context and strategic requirements. Finally, we'll

cover how the output of the digital strategy plays a pivotal role in defining the Process Inventory strategy, which is essential for determining the priority models, use cases, and resources to apply within this framework to contribute the highest value.

John Deere's Digital Transformation

John Deere's digital transformation journey showcases that not only digitally native companies can excel in adopting digital technologies, but also well-established companies can maximize business gains by successfully transitioning to the digital world.

Founded in 1837 by John Deere [1], the company started as a blacksmith shop in Grand Detour, Illinois, serving as a general repairman and manufacturer of agricultural tools. They gained recognition for pioneering the self-scouring steel plow in 1837, a revolutionary invention that revolutionized farming practices. Fast-forward to the twenty-first century, John Deere has become a global leader in the agricultural industry, with a revenue of \$52.77 billion and a workforce of 82,200 employees. By demonstrating a commitment to leveraging technology to drive innovation, enhance customer experiences, and optimize internal operations, John Deere has transformed its products and internal processes to stay ahead in the rapidly evolving landscape.

Innovations in customer products have been a hallmark of John Deere's digital transformation. They unveiled a fully autonomous tractor equipped with six pairs of stereo cameras that enable 360° obstacle detection and distance calculation. The machine continuously checks its position with GPS technology to automate movement within an inch of accuracy [2].

In 2017, John Deere acquired Silicon Valley technology company Blue River to apply artificial intelligence to agriculture. Leveraging machine learning and computer vision technology from this acquisition, they developed an autonomous "See and Spray" system, which uses cameras to identify and spray weeds with precision, reducing herbicide usage by 80–90% [3] compared to the traditional blanket spraying of chemicals.

John Deere built a digital ecosystem that empowers farmers to access and analyze farm-related data, optimize equipment utilization, and collaborate with ecosystem partners for valuable insights. They created a cloud-enabled JDLink system that connects and manages all the machines on a farm. Additionally, the AI-based data platform, John Deere Operations Center, allows customers to access farm-related data and insights. This transformation has enabled John Deere to create an ecosystem beyond just selling farm equipment, offering a suite of smart products such as farm equipment, irrigation systems, speed optimization systems, and weather data systems, all connected through Internet of Things (IoT) technologies [5]. They have

evolved their business model from being solely a products company to a services company with an additional revenue stream via subscription fees.

Furthermore, John Deere didn't limit its digital transformation to customer products but focused inward on enhancing internal operations. The company is deploying 5G in all its manufacturing facilities [6] to expand factory capabilities. This includes the implementation of edge computing, analytics, autonomous devices, and a suite of smart applications such as real-time location systems, asset tracking, inventory management, wearables, building automation, and robotics. These initiatives drive operational cost savings and contribute to the goal of creating smart, connected factories and supply chains.

Tami Hedgren, manufacturing lead for tractors and combines, highlights the significance of these efforts, stating, "With Smart Connected Factories, we will identify issues before they become problems and have solutions ready before they become disruptions. This will help make us more efficient and our customers more productive." [7]

Through these strategic initiatives, John Deere positioned itself as a leader in the digital revolution. The company's forward-looking approach aims to drive revenue growth, increase market share, and deliver superior customer experiences in the agricultural industry. By embracing digital transformation, John Deere demonstrates its commitment to remaining at the forefront of innovation and serving the evolving needs of farmers worldwide.

Digital Strategy Trends

The first step in seizing on a digital opportunity is to establish a digital strategy. A digital strategy is a comprehensive and forward-thinking plan that guides organizations on how to harness the power of digital technologies to deliver business results in the digital age. The digital strategy, at a minimum, should be aligned with the corporate strategy, and in many cases, they are one and the same.

Each organization and the industry they serve are unique, necessitating an examination of all aspects of their business model to identify vulnerabilities and opportunities that can provide a competitive advantage. The outcome of this process should be a vision for the future, detailing their digital aspirations, and a prioritized roadmap to achieve these aspirations.

For many organizations, this transformation may be disruptive as it requires challenging existing processes and cultures that reinforce a particular way of working. Embracing new practices will entail building new muscle memory in reimagined business processes, necessitating significant investments in time, effort, and financial resources. The digital strategy must inspire and motivate employees, stakeholders, and customers, providing clear direction and purpose.

The external environment is prone to change during the digital program, requiring a digital strategy that envisions a culture of innovation that is agile and adaptable, capable of capitalizing on emerging trends and technologies.

In this section, we explore the top digital trends driving transformational change and revolutionizing the way companies operate, engage with customers, and create value. Understanding these trends allows businesses to determine relevance to their context and chart their course toward a digital future.

Business Model Disruption

The digital age has brought an unprecedented level of disruption to traditional business models. This represents a threat to established businesses as smaller upstarts, and some larger innovative companies such as Apple, have leveraged technological advancements to introduce new ideas and value propositions, addressing unmet customer needs with greater convenience, efficiency, and affordability, to capture market share. In many cases, these challengers are unburdened by the complexities of outdated technology systems, complicated organizational structures, and even high regulatory requirements. Classic examples include Netflix displacing Blockbuster Video, Amazon displacing Borders Group in selling books, Uber displacing many taxi services, and Airbnb challenging existing hospitality businesses. This era also brings opportunities, not only to new digitally native startups but also to established businesses that can disrupt their own organizations with a re-envisioned business model, a culture of innovation, and the agility to transform how they operate. In this section, we'll explore some of the new business models that are emerging.

The dot-com boom that started in the mid-1990s marked the beginning of the **ecommerce business model**, which is defined as enabling customers to buy and sell goods and services over the Internet. This business model, which has challenged traditional brick-and-mortar businesses, has evolved with the maturation of smartphones, tablets, and other Internet-connected devices and apps. This area has become increasingly competitive as ecommerce is projected to account for 20.4% of global retail sales by the end of 2022, up from only 10% five years earlier [8]. A couple of recent trends have emerged which should be considered. One trend is direct-to-consumer (D2C) ecommerce, where consumer brands establish a direct sales channel to customers to form a deeper connection and maintain control of their brand experience. Successful examples of D2C ecommerce include Warby Parker for eyewear and Dollar Shave Club for shaving products, both of which have effectively bypassed traditional retailers to engage directly with their customers. Another trend is the integration of ecommerce presence with brick-and-mortar operations through an omnichannel strategy, providing customers with a seamless experience across both channels. Walmart, which

began with physical stores, has heavily invested in building its ecommerce channel and integrated the experience with features such as “Pickup Today” and “Ship to Store.”

The global subscription market size is expected to increase from \$72.91 billion in 2021 to an expected \$904.2 billion by 2026 [9]. This underscores another business model shift to that of the **subscription business model**. In the digital era, companies are finding that subscription models help organizations drive greater customer engagement from which they can gain better insights into behaviors and they are provided a predictable stream of revenue. The migration of software to the cloud has driven more software providers to move away from traditional licensing models to subscription-based models through software as a service (SaaS). In this model, customers purchase subscriptions to software hosted in the vendor’s cloud environment, which provides them a cost-effective solution that can scale on demand, and they are provided with regular software updates.

The model that may have had the most success in this era is the **platform business model**, in which a company creates a platform that connects multiple groups of users. These models allow enterprises to set up powerful ecosystems for value exchange and innovation among participants [10]. Some of the largest and most successful companies by market capitalization in this era, such as Apple, Microsoft, Amazon, Alphabet (Google), Facebook, Uber, Airbnb, and Alibaba, are either based on or prominently integrate this model. Established companies are also having success setting up platforms. General Electric, founded in 1892, has set up, through GE Digital, the Predix platform as a service (PaaS), which is an Industrial Internet of Things (IIoT) platform that provides edge to cloud connectivity, processing, analytics, and services to support their customers’ industrial applications. This provides their customers with data-driven insights, predictive maintenance solutions, and improved operational efficiency while providing GE with a recurring revenue model, access to data to drive additional innovations, and strategic positioning as a leader in IIoT.

Democratizing Innovation

The digital technology landscape is rapidly evolving, and this evolution doesn’t appear to be slowing down anytime soon. That means the success of a digital strategy is dependent on building a culture of innovation.

The traditional model of innovation relied on the top-down hierarchy or a dedicated R&D department to identify ideas and drive solutions in products, services, and business processes. Others may be constrained by their leadership, hindered by organizational silos, or lack the tools and access to information to pursue their ideas. Digitally maturing companies are finding success by sourcing innovative ideas and solutions externally through partner

ecosystems and internally by promoting and rewarding cross-functional collaboration. A recent study found that digitally innovative organizations grew their revenue more than twice as fast as those that haven't built this capability [11].

Open innovation is the practice where organizations don't solely rely on their own internal knowledge, sources, and resources for innovation but also utilize multiple external sources, such as customers, partners, and the public at large, to drive innovation [12]. A compelling example of successful open innovation is LEGO. In 2004, the company faced a crisis and was on the verge of bankruptcy due to a series of unsuccessful products that failed to excite customers, which was compounded by increasing competition from video games and online activities. To find new product ideas, LEGO turned to a different source—their customers [13]. They introduced the LEGO Ideas platform, allowing fans to upload product ideas, and if an idea received 10,000 votes, LEGO would review it for possible production [14]. This approach not only led to the creation of products that resonated with their customers but also helped build a community of loyal enthusiasts.

Democratizing innovation is expanding ideas and innovation internally. This requires building a culture of innovation and capabilities in people to explore ideas. This happens to coincide with the disruptive impact that generative AI and other advanced automation technologies will have on the workforce as routine and redundant tasks are eliminated. Organizations will have to shift the workforce toward higher cognitive activities that require critical thinking and problem-solving skills. Many employees have developed deep expertise in their areas and just need leadership to build an environment around them where they are encouraged to pursue their creativity, connect dots, and create novel solutions that generate business value. This starts with a philosophy and tone that removes hierarchical barriers, values openness and experimentation, and rewards initiative. This also means providing training, breaking down organizational silos through cross-functional teams, and giving people access to resources, tools, and data for them to understand the big picture.

A great example of getting this right is the innovation powerhouse, 3M [15]. 3M has created a wide variety of innovation centers and technical forums to facilitate building a network among scientists, employees, and customers, where they freely exchange their ideas and discuss future projects. They have implemented a 15% rule, which encourages employees to dedicate that portion of their time to innovation. This concept traces its roots back to 1914 when William McKnight, the company's general manager at the time, instituted the philosophy of "Listen to anybody with an idea" [16]. This approach has led to the development of several successful products, such as the Post-it Notes and Scotchgard. 3M has set a goal of 30/4, meaning they expect 30% of their revenue to come from products created within the last four years.

Focus on the Customer

The rapid advancement of technology has compelled organizations to embark on digital transformation journeys. However, many successful companies are shifting their perspective to focus on customer engagement as the core of their transformation strategy, with technology as a critical enabler.

This approach focuses on viewing the business from the outside-in—that is, from the customer's point of view—to redefine customer experiences, relationships, and processes. Statistics demonstrate the validity of this approach as 84% of organizations that focused on improving customer experiences increased their revenue and 79% reported significant cost savings. Deepening relationships with existing customers partly explain these stats as acquiring new customers can cost as much as five times more than retaining existing ones [17]. Nowadays, customers are heavily influenced by ratings and endorsements, with 85% reading online reviews before making a purchase, making a strong brand imperative in acquiring new customers [18].

A critical aspect of success in this endeavor is to build a companywide culture of commitment to customer centricity. This means aligning mission, values, and incentive systems to this goal. This culture fosters a continuous feedback loop, encouraging companies to listen and adapt to customer needs. Tesla, the electric vehicle pioneer, stands out for its customer-first approach, and they have been rewarded with unparalleled customer loyalty as a staggering 80% of customers buy or lease another Tesla for their next car [19]. Handling everything from production to sales in-house, Tesla maintains complete control over the customer experience, where most car companies put sales and service in the hands of third-party dealerships. Notably, Tesla's transparent communication during a previous manufacturing issue further solidified its customer relationships as Elon Musk sent personal responses to customers who tweeted at the company with honest updates on the situation.

Gaining a deep understanding of the customer's needs from their perspective is essential. A standard method for doing this is creating customer personas, which detail a customer's perspective. Clothing retailer Lululemon underscores the success of this approach. Founder Chip Wilson anchored the company's strategy on a persona he named Ocean [20]. Ocean was a 32-year-old professional woman who makes \$100,000 a year, is engaged, owns a condo, travels, is fashionable, and works out an hour and a half a day. The organization studied Ocean, who served as the inspiration for their merchandise design. This design innovation has had a transformative effect on the fashion industry, establishing them as pioneers in the emerging fashion category known as athleisure. This approach can lead to an end-to-end process view via a customer journey, in which organizations assess and improve the experience the customer has while using their products and services.

Additionally, building and maintaining data about customers to create a 360-degree view of their preferences and past interactions is essential in identifying their needs at any point in time and predicting their behavior. This is valuable in personalizing interactions, recommending products a customer might need, and identifying solutions to customer issues.

With the additional insight gained about the customer and by delighting customers and exceeding their expectations, organizations can drive a higher net promoter score (NPS), which is a standard measurement of customer loyalty and satisfaction measurement taken from asking customers how likely they are to recommend your product or service to others.

Another way to drive NPS and financial performance is to drive engagement and build a community of loyal customers by building social media channels. Nike has done this by creating a membership program that gives customers access to discounts, events, access to exclusive merchandise, and fitness communities via the Nike Training Club and Nike Run Club apps. These apps provide members access to training programs, at-home workouts, world-class trainers, wellness and nutrition guidance, and more. Through this program, they have built strategic partnerships with retailers such as Dick's Sporting Goods, which will allow customers to link rewards programs and give customers personalized shopping experiences and discounts across multiple channels [21]. The company has about 160 million active members who engage with the brand on a regular basis. This engagement, and their increased direct-to-consumer approach, has paid off for the bottom line as they announced their fourth-quarter (Q4) and annual performance for the fiscal year ending May 31, 2021, with annual profits soaring 196% [22].

Data as an Asset

The digital era has ushered in an unprecedented explosion of data. From the decade ending in 2020, the amount of data created, captured, copied, and consumed in the world increased from 1.2 trillion gigabytes to 59 trillion gigabytes, representing almost 5,000% growth [23]. With the rise of social media, online transactions, Internet of Things (IoT) devices, and other technology enhancements, the pace of data growth is only accelerating. The regulatory environment regarding data adds another layer of complexity as regulations such as GDPR (General Data Protection Regulation) in Europe, CCPA (California Consumer Privacy Act), HIPAA (Health Insurance Portability and Accountability Act), and numerous others are imposing stringent requirements on how data must be handled and protected. This is leading the management and governance of data to become a high priority for many organizations.

Data can be a source of competitive advantage for organizations that recognize its value as an asset. Those who can effectively manage and harvest it to gain

valuable insights and make stronger decisions will gain an edge. Data not only helps companies understand their customers better, as was just covered, but it's critical in the adoption of new technologies such as machine learning, and it can fuel innovation in products, processes, and business models. That is why developing a strategy for the use and management of data must be a consideration for a digital strategy.

This explosion of data combines with the emergence of AI capabilities, leading organizations to build stronger data analytic capabilities where decisions are made with empirical information rather than gut feeling. This starts with the ability to gather data from multiple input sources with quality that decision-makers can trust. To unlock this potential, advanced analytical models need to be created to identify patterns, predict trends, and optimize decision-making. To be effective, these models need to work backward by first identifying the business outcome to be optimized. This may require a culture shift where leaders embrace this new approach and decision-makers in the business partner with data scientists. An excellent example of this strategy is Walmart, the world's largest retailer, which has fully embraced a data-driven approach with AI and machine learning to optimize its supply chain and deliver enhanced customer experiences. With hundreds of in-house data scientists dedicated to various aspects of the business, Walmart's cutting-edge AI technology has evolved from predicting sales demand to understanding and fulfilling customer preferences. Through advanced robotics, machine learning, and predictive analytics, Walmart's supply chain is becoming more efficient, enabling next- or two-day shipping to 95% of the US population and increasing inventory accuracy [24].

As organizations conduct their regular business operations and accumulate vast amounts of data, some are recognizing the unique value their data holds for other entities. This realization has led to the creation of new revenue streams as they monetize their data. Mastercard is a great example of this, with about 3 billion cardholders making approximately 90 billion transactions yearly at around 70 million merchants through about 40,000 banks and financial institutions in about 210 countries and territories [25]. This represents valuable insights into the purchasing trends of consumers that consumer-facing businesses can leverage to develop a holistic view of different customer segments. Mastercard established the Data & Services unit, which leverages anonymized and aggregated transaction data to offer clients a portfolio of services, including reports, data products, analytics solutions, and consulting services. Monetizing data has become a significant source of revenue for Mastercard and added value for their clients.

Defining a digital strategy that extracts maximum value from data must start with a commitment from the top. Many organizations are hiring a Chief Data Officer, who reports directly to the CEO. The chief data officer, or an equivalent senior-level data leader, will be charged with creating a data

strategy. This strategy must lead with business objectives and include the definition of a data management program that consists of data governance and quality processes, data privacy and compliance policies, training and cultural alignment, a data architecture vision, and the required data and analytics capabilities to enable data-driven decision-making.

Defining the Digital Program

As each organization and the industry they serve are distinct, defining a digital strategy and executing change through a digital program will require organizations to critically assess their own threats and opportunities to identify the competitive advantage they need to pursue. The digital trends that I just covered, while not exhaustive, highlight what must be considered when doing this assessment.

Sunil Gupta, a professor at Harvard Business School and author of the book *Driving Digital Strategy*, highlights the keys to success in this journey as “four things: One is your business strategy; the second is about your operations and value chain; third is about how you engage with customers; and fourth is how you structure the entire organization” [26].

The approach in Figure 3-1 represents how each of these keys can be considered in a structured manner. While this appears sequential, most times it’s executed iteratively as organizations test new business models and improve strategies and designs based on those insights.

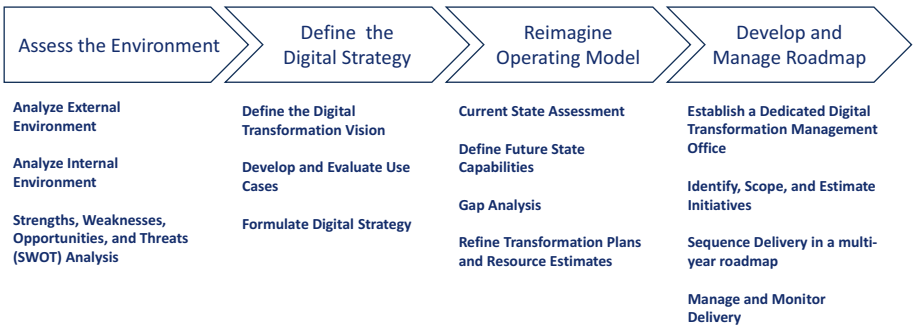


Figure 3-1. Defining the Digital Program

Assess the Environment

In this step, insights are gathered and analyzed from both external and internal sources to identify opportunities and threats, as well as assess the strengths and weaknesses that need to be addressed during the transformation.

Analyzing the external environment can be facilitated through the PESTEL framework and the Five Forces analysis. The PESTEL framework offers valuable insights into the macroenvironmental influences that impact businesses, encompassing six key factors: political, economic, social, technological, environmental, and legal [27]. The Porter's Five Forces covers the competitive forces of competitive rivalry, supplier power, buyer power, threat of substitution, and threat of new entry [28].

I will highlight how some of these forces may impact digital strategy:

- **Political influences** pertain to potential laws and regulations surrounding new technologies. We are seeing that play out right now with debates in Washington, DC, on how to regulate artificial intelligence.
- **Social influences** are currently shaping customer behavior, especially among tech-savvy younger generations, underscoring the need to align digital strategies with evolving preferences.
- **Legal considerations** entail compliance with existing laws and regulations, which hold critical implications for digital strategies. In some industries, the regulatory environment is so onerous that it creates a barrier to entry for digitally native startups. This is the case for Fintech companies that build technology to enable digital access to financial services but can't directly compete with traditional banks due to the prohibitive cost of compliance.
- **Competitive rivalry** is where you assess the strengths and weaknesses of your competition in your industry. For established players, this could include threats from digitally native startups.
- **Buyer power** is evaluating the power of your customers to switch to competitors, which would give them greater bargaining power in establishing pricing.

Performing an internal analysis requires leaders to honestly assess the strengths and weaknesses of internal capabilities to adopt a digital agenda. For established companies, there are many strengths, such as the quality of their assets, the loyalty of existing customers, and the expertise of their people. However, several weaknesses can exist if the culture isn't aligned with agile ways of working, the organizational structure is siloed and complex, data isn't managed effectively, and technical complexity becomes burdensome.

Internal and external analyses are summarized in a SWOT analysis, which categorizes the organization's strategic position internally as strengths and weaknesses and externally as opportunities and threats. This comprehensive analysis provides leaders with the needed information from which to develop the digital vision by defining for them where to focus their digital energies.

Define the Digital Strategy

When defining the digital strategy, it's essential to be bold and set an ambitious tone throughout the process. Advancements in technology and changing customer expectations are leading innovative companies to disrupt traditional business models and displace previous market leaders. Some new business models are subject to a network effect that creates a winner-take-all situation for first movers and fast followers [29].

This leads many organizations to adopt an “art of the possible” approach, which means that idea generation is not bound by the status quo, existing constraints, and preconceived notions. This fosters creative thinking and innovation, which is the key to discovering groundbreaking ideas that have the potential to revolutionize the organization and their industry. The result is a broader set of ideas that can later be evaluated and refined based on feasibility and potential impact.

According to BCG, “Only 33% of organizations successfully meet the challenge of digital disruption. Those that do tend to make big strategic bets—on the order of 10% of their market capitalization [30].” Winning digital strategies aim to move the needle in terms of value creation.

The key to maintaining alignment is to have your digital strategy anchored in a strong definition of “Why” as expressed in the **digital vision statement**. The vision statement should clearly state the objectives of the transformation and communicate the expected outcome when it's complete. The vision statement should inspire stakeholders to act and facilitate decision-making along the transformation journey.

As part of this process, it's important to identify the value that is expected for the organization via high-level digital transformation goals. Figure 3-2 illustrates some sample goals that can be adopted. Identifying goals will be important for evaluating and prioritizing transformation options and use cases as well as defining success criteria for the program.



Figure 3-2. Digital Transformation Goals

The next step in this process is to develop transformation use cases, which are specific scenarios that the organization aims to implement or improve using digital technologies and solutions. Use cases are essential in evaluating, planning, and executing specific digital priorities. Broadly, there are two categories of use cases: strategic use cases and operational use cases.

Strategic use cases are focused on innovating offerings, services, and products that a business provides to customers to generate revenue. On the other hand, operational use cases are focused internally on the operating model, which includes culture, employee talent, supporting processes, partnerships, use of technology, and data to support the generation of revenue.

Each of the proposed use cases is detailed with a description, objectives, value proposition, and benefits. Leadership will assess each of the use cases based on the benefits for the business to determine priority for implementation.

The culmination of this step is to summarize the digital strategy in a document that covers the vision, goals, initiatives, key priorities, and other important information. This strategy, which is essential in defining how to win in this agenda, can be shared broadly with stakeholders to build support and collaboration when starting the program.

Reimagine Operating Model

Executing an ambitious digital transformation strategy can have profound implications for how legacy organizations operate. It requires not only leveraging digital technologies to maximize business value but also cultivating organizational agility and adaptiveness to respond promptly to changes in the external environment. As shown in Figure 3-3, the operating model serves as the backbone of the organization, where various components, such as resources, capabilities, processes, and technologies, come together to deliver on its mission and objectives.

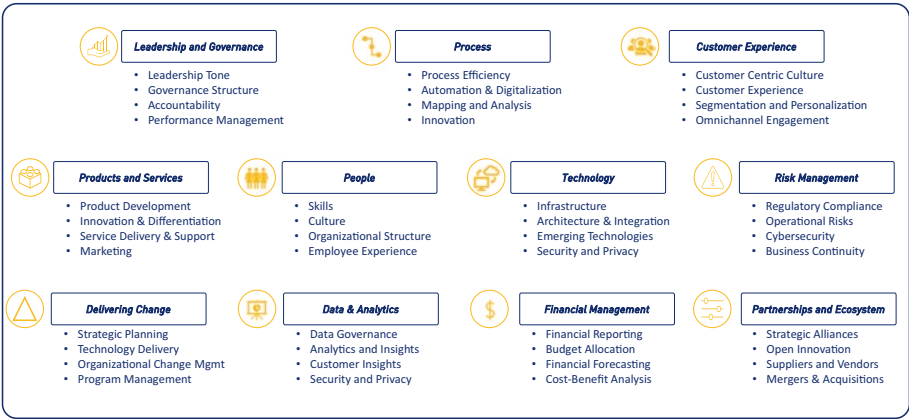


Figure 3-3. Operating Model Components

There is an intricate interplay between the components in Figure 3-3 which has a crucial role in enabling the organization to achieve its goals. For instance, the right combination of people and talent, supported by effective processes and technologies, is essential to drive innovation and adapt to rapidly changing market conditions. Similarly, a well-structured governance and decision-making framework ensures that the digital transformation initiatives align with strategic objectives and are monitored for progress.

In many organizations, the operating model evolves naturally as the business evolves. While this may work fine in regular circumstances, it presents challenges when executing on a transformational agenda. Hence, proactively defining and articulating the future state operating model becomes paramount to aligning the digital strategy with day-to-day execution.

By explicitly identifying the connections between the various components of the operating model and their roles in supporting digital transformation, organizations can better understand the intricacies involved. This understanding empowers them to identify potential bottlenecks, anticipate challenges, and align their transformation initiatives more effectively to drive successful outcomes.

Performing a current state assessment, by leveraging this framework, ensures that all existing components are considered, reveals the strengths and weaknesses of the organization, identifies areas for improvement, and serves as a starting point for the transformational journey.

Defining a future state operating model that aligns with the digital vision and identifying gaps that need to be addressed are pivotal steps in bridging the connection between the current operating model's complexity and the digital transformation journey's success. Together, these activities provide a clear

pathway for organizations to navigate the complexities and challenges associated with transforming into a digitally powered enterprise. It also enables teams and functions across the organization to understand more clearly how their efforts support success in the transformation.

Develop and Manage Transformation Roadmap

Executing on a digital transformation journey is a significant undertaking that demands leadership commitment, thoughtful planning, and rigorous execution. These programs require large investments and multiyear coordination across disparate business and functional units, which must balance the needs of ongoing business operations. Dependencies must be managed, and the inevitable challenges will have to be overcome. Without a solid execution plan, even the strongest digital strategies will result in missed expectations, frustrated stakeholders, and cost overruns. In this phase, I'll cover some of the best practices that organizations are implementing to deliver on their digital agenda.

Transformation efforts require versatile leadership to navigate disparate stakeholders, change the capability and culture of the organization, and keep everyone focused on the transformational goals. This is leading to the emergence of the chief transformation officer (CTO) or a similar role that ideally reports to the CEO or, at a minimum, has a seat at the leadership table, signaling to the organization that top management is serious about the transformation and committed to its success [31]. An effective CTO typically has broad experience, especially across business and technology domains, and they have earned trust and credibility across the organization, which is critical in mediating the inevitable organizational friction. They must be capable of driving the big picture toward the strategic vision while also rolling up their sleeves to direct the details of the program.

CTOs will establish and lead Transformation Management Offices (TMO). TMOs provide a central point of planning and coordination across all digital initiatives. They are responsible for establishing consistent processes, accountability, frameworks, and tools that govern the effort with transparent reporting of risks, milestones, and benefits being gained.

A critical action for the TMO is to create and maintain the execution roadmap. This involves leveraging transformation use cases and future state operating model designs to identify specific initiatives with logical scope that can be executed as a unit. Detailed planning for each initiative is undertaken to refine scope, identify dependencies, and determine resource requirements across functional areas. This team will coordinate project planning, identifying resource requirements across each type, and determining projected timelines and costs for each initiative.

Identifying the right talent for success is a crucial part of this process as many organizations may not have sufficient capacity or capability to drive such a large effort. Organizations will need to build internal capabilities through targeted training and hiring talent externally. In many cases, partnerships with external service providers can bridge the talent gap, provide additional capacity, and bring in fresh perspectives and expertise.

Sequencing initiatives in a multiyear roadmap requires a deep understanding of dependencies and balancing between quick wins that demonstrate results with foundational priorities, such as a new enterprise data platform, which requires a greater duration but is necessary to deliver on some big bets.

The chief transformation officer and the Transformation Management Offices must instill a culture of delivery and build monitoring processes that provide transparency on the status of efforts and their progress toward their goals so that, if needed, leadership can take corrective actions.

Establishing a Process Inventory Strategy

As established in Chapter 1, a key to success in transforming the organization is gaining alignment with processes through the Process Inventory framework. This is core to rebuilding the capabilities of the organization as the vertical and horizontal alignment that Process Inventory will enable is essential for enterprise agility.

To capitalize on this promise, it's crucial to thoughtfully create a strategy for implementing the Process Inventory framework. Creating required models, aligning metadata, and implementing use cases, such as those covered in Part 2 of this book, will be a multiyear effort. A winning approach is to start small, perhaps with a pilot in a single business unit, to demonstrate the value of this approach, and then scale it more broadly after the model is proven.

The best place to start is with the output of the digital strategy as it will detail the goals, use cases, and operating model impacts in the transformation. For instance, if a priority goal is to improve customer engagement, then that would mean creating a customer experience capability that builds and analyzes customer personas and customer journeys and aligning it with Process Inventory. If the goal is to increase operating efficiency, then detailing process flows with the resources leveraged may be a priority.

A business case for this framework and the resources to enable it will be required to gain alignment across leadership, which means showing how this directly supports the digital strategy will increase the chances of obtaining needed financial and stakeholder support.

The business case, like other initiatives seeking funding, needs to detail the plan, process use cases, points of collaboration, the resource plan, and the financials to get started. This should be integrated into the digital roadmap that the TMO is managing so that process dependencies across other digital initiatives can be identified and tracked.

Key Takeaways

- **Defining a digital strategy** is an important step to clarify the goals of the transformation, which is critical for aligning all efforts with those goals.
- **Key trends** are emerging in this digital era that are reshaping business models and transforming how organizations operate.
- **Creating a digital program** requires a meticulous approach that considers all opportunities and implications in defining and managing an optimal transformation roadmap.
- **Articulating a Process Inventory strategy** is important for defining the priority models, use cases, and resources necessary to apply this framework and contribute the highest value.

In Part 2 of this book, we'll explore how to apply the Process Inventory framework to transform organizational operations. This exploration begins with the next chapter, which focuses on leveraging this framework to drive operational excellence.

PART

II

Applying Process Inventory to Transform Ways of Working

Driving Operational Excellence

This chapter explores several key aspects that contribute to achieving operational excellence. We'll start by defining operational excellence as the driving force behind comprehensive operational improvement efforts. Then, we'll delve into the Lean Six Sigma methodology, highlighting its role in process optimization. Additionally, we'll detail how the Process Inventory directly enhances the effectiveness of Lean Six Sigma initiatives.

Operational excellence extends beyond process improvement, so we'll also explore several other critical programs that are essential for achieving organization-wide operational excellence. Throughout, we'll emphasize the pivotal role that the Process Inventory plays in these efforts.

The Impact of the Toyota Way

The story of Toyota's operational excellence and advancement of the Lean methodology is one deeply rooted in history, philosophy, challenges, and remarkable achievements. The journey of Toyota's success, as driven by the

Toyota Way and the Toyota Production System (TPS), serves as a remarkable case study of how a company's core principles can transform the automotive industry and business practices globally [1].

Toyota was established in 1937 by Sakichi Toyoda and his son Kiichiro Toyoda. The company's journey toward operational excellence finds its origins in the ideals of Sakichi Toyoda. He was not only an engineer but also a visionary whose philosophy emphasized continuous improvement and dedication to contributing to the greater good. This philosophy is distilled into the “five main principles” of adhering faithfully to responsibilities, pursuing knowledge and innovation, practicing practicality, fostering a welcoming work environment, and upholding respect for spiritual values while maintaining gratitude. These principles guide Toyota's actions even today.

Post-World War II, Toyota faced unique challenges, starkly different from those of American automakers. While competitors, such as Ford, embraced mass production, economies of scale, and big equipment to produce large volumes of parts as cheaply as possible, Toyota couldn't follow this model as the Japanese market was much smaller and they didn't have sufficient cash on hand. This challenge led to a crucial realization: shorter lead times and flexible production lines were needed to manufacture diverse vehicle models on a single assembly line, fostering a culture of waste elimination and quick cash turnover to pay suppliers.

Kiichiro Toyoda's vision for the Koromo plant laid the groundwork for the revolutionary Kanban system. Initially documented using slips of paper, this system aimed to produce required parts each day, necessitating flow production across every operation. Quality became paramount as Toyota recognized that JIT (Just-in-Time) required impeccable quality at each step. Standardized work, JIT, and the development of the Kanban system formed the backbone of the Toyota Way. The Toyota Production System (TPS) emphasized “getting the facts” and identifying root causes—a scientific mindset that continues to drive Toyota's success.

Toyota's adherence to the Toyota Way yielded remarkable results. The company consistently delivered top-quality vehicles, witnessed steady growth in sales, and achieved consistent profitability. Toyota built massive cash reserves, which further funded innovation, contributing to society and local communities. Financially, Toyota's net profits outshone competitors by a staggering margin; they were the world's first automobile manufacturer to produce more than ten million vehicles per year, a record set in 2012 [2], underscoring the potency of their philosophy.

The story of Toyota's journey to operational excellence is one of continuous evolution rooted in a distinct philosophy. The Toyoda family's ethos—embracing challenges, innovation, and contributing to society—served as a guiding light. Toyota's paradigm-shifting approach, with lean principles at its core, heralded

the decline of traditional mass production and ushered in a new era of lean production practices. The Toyota Way stands as a testament to the power of principled thinking and a relentless pursuit of excellence. From its origins to its operational achievements, Toyota's journey resonates as an inspiring case study of how a commitment to innovation, pragmatism, and continuous improvement can pave the way for enduring success in a rapidly evolving world.

Organizational Inefficiencies and Complexity

British naval historian and author Cyril Northcote Parkinson published an essay in the *Economist* in 1955, where he detailed his observations of the British navy's evolution from 1914 to 1928. This essay came to be known as "Parkinson's Law" found that despite a reduction of navy ships by two-thirds and personnel by a third during those years, he noted that the number of bureaucrats had increased by almost 6% annually [3]. The observation was clear: fewer personnel and reduced workload, yet the administrative ranks continued to expand. Parkinson's opening line poignantly stated: "It is a commonplace observation that work expands so as to fill the time available for its completion." In bureaucratic organizations, headcount tends to expand to consume the available budget. Parkinson identified two forces at play: the "multiplication of subordinates," wherein officials prefer subordinates to rivals, and the phenomenon of officials creating work for one another. Subsequent studies further explored the repercussions of this surplus headcount, revealing that in larger group settings, unanimous decisions can become elusive as subgroups form to block one another.

Along with these complexities, many organizations face additional challenges with system proliferation, often resulting in overlapping functionality and stifling IT spending. Complex vendor relationships, an explosion in the growth of data, increasing risks, and evolving regulatory requirements further compound challenges to operational efficiency.

In this context, costs and headcount become bloated. More importantly, getting things done becomes challenging, which doesn't bode well when attempting a change as significant as digital transformation. Changes can become time-consuming and expensive, or worse, encounter resistance as teams undermine efforts to protect their turf. It becomes almost impossible for leadership to identify which resources are essential for the organization.

Significant investments have been made in efficiency efforts, utilizing both internal resources and external consultants, with the goal of identifying and reducing waste and inefficiencies. Despite these investments, the collective benefits often fail to materialize, often due to a narrow focus on specific processes or business areas. The absence of a broad, enterprise-wide perspective can lead to a situation where efficiencies gained in one area are offset by waste accumulating in another.

Operational Excellence

As a response to this challenge, a comprehensive concept known as operational excellence has emerged, which strives to achieve optimal performance across the organization.

Operational excellence is a state in which the organization is achieving and maintaining the highest level of performance aligned with its vision and goals. While this state may not be achievable as there is always room for improvement, this defines a philosophy that directs efforts toward creating a purpose-driven culture of continuous improvement throughout the organization. This enables efficient resource utilization and the elimination of waste to deliver a sustainable competitive advantage. This program empowers organizational agility in the face of change while maximizing overall company performance and creating exceptional value for customers and stakeholders.

Keys to Operational Excellence

The starting point for driving operational excellence is to be clear about what that means in the context of your organization. This requires leadership to articulate a **vision for the future that identifies and prioritizes the goals** that are important in operating the organization. Figure 4-1 illustrates sample goals for consideration in this exercise. Furthermore, a definition of the aspirational or ideal state design can be created, which outlines key design concepts to focus on, but there should be some recognition that this is a long-term journey, and this design will likely evolve and adapt as lessons are learned and as the external environment changes. The importance of this future state vision and goals is to define and serve as an important “why” or purpose that will motivate internal and external stakeholders and ensure that every effort and action is aligned toward this shared purpose.



Figure 4-1. Sample Operational Excellence Goals

The **scope of this effort should be enterprise-wide**, or for large and complex organizations, a significant business unit. Traditionally, many improvement efforts have been focused on a small set of processes to identify

inefficiencies and implementing local solutions. This fails to recognize that organizations are composed of multiple functional domains and customer touchpoints cut across multiple business areas, which means that local solutions may address that problem but will have minimal impact on the overall operating environment. Figure 4-2 illustrates the various components of the operating model, which when considered holistically can drive needed synergies across these functional domains, remove silos, foster collaboration, and propel a higher level of operational performance.

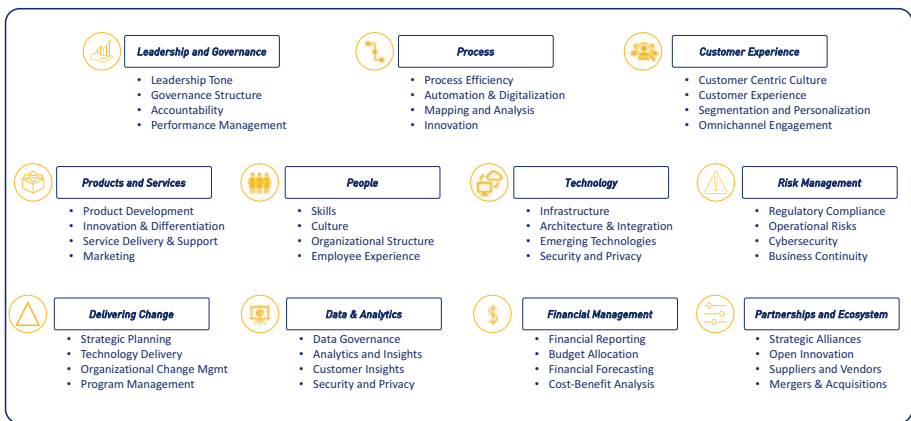


Figure 4-2. Operating Model Components

Operational excellence **programs can vary in scope and size**. They can be sizable programs with a significant multiyear investment, involving numerous workstreams, or they can be less formal and driven by a cultural shift toward incremental improvements through methods such as Kaizen events—short cross-functional workshops focused on a targeted area or process. Regardless, a program structure, perhaps through a Program Management Office, should be set up to ensure that all efforts are aligned to the future state vision. They should define success and establish key performance indicator (KPIs) targets to measure progress. KPIs should be collected and published regularly to provide insights into the effectiveness of improvement efforts.

A program of this nature needs to have **strong leadership** that communicates the vision, progress, and benefits to the entire organization. Consistency and commitment are key; leaders must walk the talk and demonstrate through their actions that they are modeling their behavior and being accountable for the success of this endeavor.

Leadership must **cultivate a culture of continuous improvement**. People, at all levels of the organization, are the engine that will drive operational excellence as they have developed deep expertise in their domain. The emphasis on critical thinking and questioning every assumption will lead to

valuable insights and perspectives as to how to optimize processes. In the Toyota Way, people were at the center of the philosophy, and they were expected to spend a lifetime working to perfect their craft [1].

In traditional models, innovation was driven exclusively by the hierarchical structure or specialized efficiency teams. However, in the **democratic innovation** model, employee engagement and contributions must be encouraged. Leadership needs to remove barriers, arm employees with tools and data, enable cross-functional collaboration, and set the tone where everyone can pursue their innovative ideas. Failed ideas will inevitably occur, but this should be viewed as a learning opportunity, and the reward system should value openness and recognize employee initiative.

This will lead to the side effect of an engaged and high-performing workforce. According to Herzberg's two-factor theory [4], employee satisfaction can only be negatively impacted by not providing sufficient salary, job security, status, etc. Positive motivating factors such as challenging work, achievement, recognition, and opportunities drive greater engagement. In other words, **employees seek empowerment to find purpose in their work**. Empowering employees to make decisions and contribute to process improvements enhances their sense of ownership and accountability.

Operational Excellence Outcome

Organizations that have been successful in their pursuit of operational excellence are driven by a strong purpose, which empowers motivated employees to deliver superior customer service and value for shareholders. The organization operates with minimal waste, produces high-quality outputs, optimizes processes, and mitigates risk. This increased efficiency provides the agility needed to respond rapidly to changing competitive and environmental pressures.

Lean Six Sigma

While operational excellence provides the holistic vision, purpose-driven culture, and program for achieving a high-performing operating environment, Lean Six Sigma offers a rigorous and structured methodology, a set of tools, and techniques to aid in managing and improving quality and performance, as well as solving potentially complex problems to enhance processes. The term "Lean Six Sigma," an integration of two distinct methods, was introduced in the 2001 book *Leaning into Six Sigma: The Path to Integration of Lean Enterprise and Six Sigma* [5].

Lean, as a term, was conceived by John Krafcik, a researcher at MIT, while studying the performance attributes of the Toyota Production System [6]. He commented at the time, "It needs less of everything to create a given amount

of value, so let's call it Lean." The world of Lean thinking grew rapidly, focusing on reducing non-value adding activities or waste, known as "muda" in Japanese. The Lean approach is truly about understanding how the work gets done, finding ways to do it better, smoother, and faster, and closing the time gap between the start and end points of processes.

Six Sigma is a highly disciplined approach focused on delivering near-perfect products and services by minimizing the variants within a process and identifying the causes of defects. Its roots trace back to the US in the 1980s at Motorola, during a time when they were struggling to compete with foreign manufacturers. They set a goal of achieving tenfold improvement in five years, centered on a plan for global competitiveness, participative management, quality improvement, and training. The term "Six Sigma" was coined by Motorola quality engineer Bill Smith, signifying an improvement measurement derived from a statistical term denoting three or four defects per million.

Six Sigma was further popularized by General Electric in the 1980s after Jack Welch, the dynamic CEO at that time, became a champion of the cause. After an initial successful pilot, it became evident that Six Sigma projects could yield attractive financial benefits. Welch mandated the use of Six Sigma across the entire group of GE businesses [7], stating, "We are going to shift the paradigm from fixing products to fixing and developing processes, so they produce nothing but perfection or something very close to it."

Lean and Six Sigma complement each other perfectly as central to both methods is the recognition that "process" is what needs to be improved. Six Sigma enhances the efficiency-driven Lean considerably through its strong roots and toolkits for measurement and data analysis. Lean Six Sigma (LSS) is a powerful approach for delivering necessary process improvements to drive operational excellence.

Lean Six Sigma Principles

Lean Six Sigma has a foundation of seven core principles, which represent the essence of the methodology. We'll delve into these principles as it's critical to understanding how Lean Six Sigma drives continuous improvement.

Principle #1: Focus on the Customer

To deliver the best customer experience, it's essential to understand your customers' wants and needs. The aspects of your service or offering that customers consider most crucial are referred to as CTQs (critical-to-quality) requirements in Lean Six Sigma. CTQs serve as the foundation for defining metrics that assess meaningful improvements and measure the effectiveness of those improvements.

Since Lean Six Sigma is focused on the value stream—the end-to-end process from initial conception through to delivering value to an end customer or stakeholder—it's important to recognize that value streams can encompass multiple internal stages involving various suppliers. In the context of Lean Six Sigma analysis, customers are defined as anyone who receives the output of a process. This definition includes both internal stakeholders and external customers.

To establish the boundaries of a process, a SIPOC diagram is created, documenting the following components: suppliers (providers of inputs), inputs (required resources), process (the actual steps), outputs (results or products), and customers (recipients of the outputs). This high-level tool helps identify stakeholders, segment customers, and centralize conversations around a unified process definition.

Understanding customer needs is achieved through a method known as Voice of the Customer (VOC). This involves interacting with them using techniques like interviews, focus groups, performing observations, and surveys. By actively listening to your customers, you can gain insights into their needs, preferences, and pain points. The insights derived from VOC sessions facilitate the development of critical-to-quality (CTQ) requirements. These requirements transform customer need statements into specific product or service requirements.

Requirements are refined further through Kano analysis, which assesses the value that customers place on features of a product or service. This process ensures that the most impactful requirements are given priority. Subsequent analysis is carried out to determine how each requirement could be quantified using metrics or measures. These quantifications form the foundation for establishing targets and identifying methods for metric collection.

Principle #2: Identify and Understand How Work Gets Done

The Japanese word “gemba,” translating to the “actual place,” refers to where the action takes place. The central concept of this principle emphasizes that Lean Six Sigma practitioners should engage stakeholders directly in the work environment. This immersion allows them to gain an appreciation of how processes are executed. Through documentation and in-depth analysis, practitioners aim to comprehend the challenges and identify opportunities for improvement.

A fundamental aspect of comprehending work involves process mapping, which serves to confirm understanding, facilitate stakeholder discussions, and provide a foundation for analysis. Various models can be constructed, tailored to the desired level of granularity and the focus of the study.

A value stream map documents the end-to-end process, including stages, activities, and handoffs as it crosses organizational boundaries to deliver customer value. This higher-level representation is designed to cover both value-adding activities and non-value-adding activities that introduce waste and inefficiency.

Process models are constructed to document how high-level activities in a value stream map are executed, including specific actors, activities, cross-functional handoffs, and leveraged resources.

A spaghetti diagram represents the flow of people, materials, and equipment in a physical space, with the intent of identifying unnecessary movement in the process.

This principle extends beyond documenting processes to include several analysis techniques. During observations, the analysis team should identify and collect relevant performance data such as the overall timeline, number of people involved, and wait times and document those on the process maps themselves. Moments of truth should be noted, which are points of interaction with external customers as they hold a higher significance in customer satisfaction or dissatisfaction. Finally, an exercise called process stapling is conducted to following the process, not from the perspective of the person operating the process but from the perspective of the thing going through the process to understand the flow to identify bottlenecks, idle time, or any other inefficiency.

Principle #3: Manage, Improve, and Smooth the Process Flow

The focal point of this principle is the identification and resolution of bottlenecks, which are process steps lacking the capacity to meet demand, and constraints, which impede the overall performance of the end-to-end process. Employing the theory of constraints, a standardized methodology is used to detect and address these limitations. This can involve subordinating other process steps to align with the constrained step, thereby preventing overproduction, or elevating the constraint itself by enhancing its capacity. The approach is an iterative one, involving the continuous identification and resolution of constraints until the end-to-end process achieves satisfactory performance.

Principle #4: Remove Non-value-added Steps and Waste

This principle focuses on streamlining processes by identifying and eliminating activities that do not directly contribute to customer satisfaction while also addressing various types of waste that can hinder agility, efficiency, cost savings, and quality.

Value, within the context of Lean Six Sigma, pertains to any activity or process step that directly contributes to fulfilling customer needs and requirements. To evaluate the value of each step, three general criteria can be leveraged:

- Does the outcome matter to the customer?
- Does the activity either physically alter the product or service, or is it indispensable for a subsequent step?
- Is the activity executed correctly the first time?

While these criteria serve as a starting point, they are somewhat vague and require interpretation. Therefore, it is imperative for each organization to establish its own definition that aligns with its specific context.

To further guide the identification of non-value-added steps and waste, Taiichi Ohno, a former Toyota executive and often referred to as the “father of the Toyota Production System,” developed a list of seven wastes applicable to manufacturing processes. Over time, this list expanded to encompass eight wastes applicable to various types of processes across different organizations:

- **Transportation:** Unnecessary movement of materials or products that can lead to waiting times, increased resource consumption, and potential loss or damage.
- **Inventory:** Excessive levels of raw materials, work in progress, or finished goods that contribute to longer lead times, obsolescence, higher costs, and delays.
- **Motion:** Wasted efforts due to unnecessary physical movement, such as bending, twisting, or walking, often caused by suboptimal equipment placement or information access.
- **Waiting:** Delays caused by people being unable to proceed with their work, leading to idle time and inefficiencies.
- **Overproduction:** Generating more products or services than needed, resulting in overstaffing, unnecessary storage and transportation costs, and excess inventory.

- **Overprocessing:** Executing more steps than necessary, consuming resources without adding value.
- **Defects:** Rework or errors caused by failing to meet critical-to-quality (CTQ) requirements.
- **Skills:** Underutilizing the skills and potential of people within the process.

The analysis of removing non-value-added steps begins with process models, where non-value-add and waste are documented based on established criteria. Capturing the processing time or, if possible, estimating the cost of each step is crucial for constructing a business case, particularly when the elimination of these steps requires an investment.

Principle #5: Manage by Fact and Reduce Variation

At the core of this principle lies the emphasis on making decisions based on factual data, rather than assumptions or intuition, to achieve process consistency. Uncontrolled variations in processes can lead to quality issues, so it's in an organization's best interest to achieve a consistent, reliable, and predictable level of performance. This means collecting data, calculating for variation, and interpreting calculations correctly to determine if actions are required.

Standard deviation is a calculation method for assessing process variation. It quantifies the extent to which individual data points deviate from the average or mean. A smaller standard deviation indicates that data points cluster tightly around the mean, highlighting consistent and predictable outcomes. Conversely, a larger standard deviation suggests greater data spread, indicating higher process variability and less predictable results. Investigating the nature of variation, whether common (expected) or special (unexpected), will determine if further root cause analysis is needed.

Control charts play a pivotal role in identifying and differentiating between common and special cause variations. These charts depict the process average and upper and lower control limits (UCL and LCL). Placed at plus and minus three standard deviations from the process average, these limits serve as reference points for assessing process stability.

While this topic can be explored further, including additional chart types like histograms and Pareto charts, the key takeaway is that calculating and understanding variation is essential for effectively managing variations and delivering high-quality processes.

Principle #6: Involve and Equip the People in the Process

This principle recognizes that the success of any improvement initiative hinges on involving the people in the process, equipping, and empowering them to challenge and enhance the way work is executed.

Achieving results thrives on collaboration rather than isolated efforts. A collaborative approach engages every organizational level. Employees, with their valuable insights into processes, hold the keys to understanding nuances and overcoming challenges. Their active participation in process improvement discussions fosters a sense of ownership and accountability. Empowered individuals become the driving force behind innovation, propelling solutions aligned with organizational objectives.

Empowering employees to identify and resolve problems not only boosts morale but also nurtures a culture of continuous improvement. Empowered individuals become change agents who inspire others. This culture embraces challenges and perceives them as opportunities for learning and growth, aligning perfectly with a growth mindset.

Equipping employees with the necessary tools and skills is critical. Offering training and development initiatives positions individuals for success in process improvement endeavors. By honing problem-solving and analytical skills, organizations cultivate a workforce adept at identifying inefficiencies, addressing challenges, and implementing sustainable solutions.

This culture thrives on transparent communication, robust feedback loops, and recognition of achievements and initiatives. A shared commitment to ongoing enhancement propels the organization forward, fostering adaptability in the face of evolving challenges.

Principle #7: Undertake Improvement Activity in a Systematic Way

This Lean Six Sigma principle emphasizes the importance of a structured, systematic problem-solving methodology for conducting improvement efforts to enhance existing processes. This methodology is best used when confronted with complex problems where causes and solutions are not readily apparent and when collaboration from individuals with diverse expertise is required.



Figure 4-3. DMAIC Framework

A critical factor in the success of a DMAIC (Define, Measure, Analyze, Improve, and Control) project is to clearly identify and describe the problem, purpose, and objectives through an improvement charter, which marks the start of the effort. The improvement charter includes the following components: a compelling business case, a concise problem statement, measurable goals, a well-defined scope, critical-to-quality (CTQ) factors, roles, milestones, and more. This charter, important for gaining alignment from sponsors and team members, should be considered a living document that evolves as the team gains a deeper understanding of the problem.

A great way to validate the quality of the improvement charter is through the 5Ws (and 1H) tool. This means ensuring that it clearly addresses the what, why, when, how, where, and who aspects in a manner that all stakeholders can easily understand.

Maintaining a pulse on progress through informal reviews with sponsors and formal tollgate reviews at the conclusion of each DMAIC phase ensures that the initiative remains aligned with its original objectives. Tollgates play a crucial role in validating the proper completion of the latest phase and the production of high-quality outputs. They also offer an excellent opportunity to revisit the improvement charter based on new insights, evaluate lessons learned, assess team dynamics, review milestones, validate assumptions, and reexamine goals.

Process Inventory Enhances Lean Six Sigma Effectiveness

The main criticism of Lean Six Sigma programs is that management adopts the methods and tools, trains their people, and expects amazing results which never materialize. They fail to realize that this success requires an enterprise-wide culture change toward a philosophy of critical thinking and continuous improvement. While there is no substitute for leadership, Process Inventory can significantly increase the effectiveness of process improvement efforts.

Sustained Knowledge Management

Optimizing processes is an iterative pursuit that entails experimenting, analyzing results, designing further solutions, and repeating these steps. This process can take place over the span of several years or more, with potentially different teams performing the analysis. Without maintaining process documentation, each team may define the problem uniquely and map the process with different boundaries. I've personally seen this many times where each new team comes in and starts their analysis with a blank sheet of paper. This not only inefficiently uses time but can also lead to poor results as past lessons and failures don't serve as a guide for future solutions.

Process Inventory, and the information governed in the Integrated Operational Repository, solves this by providing an authoritative source of process information within the environment. This provides analysis teams with an inventory of processes, their boundaries, identification of process ownership, and if process models have been created, process activities and connections upstream and downstream, as well as all the resource and operational information captured in the metadata.

If the Integrated Operational Repository is populated, then the Process Inventory should be leveraged to define scope with precision. Analysis should begin with this information, and stakeholder discussions should focus on identifying information gaps and brainstorming solutions.

If this process area has not been documented yet, then new analysis efforts represent an opportunity to create proper documentation. This means working with the Process COE to create the Process Inventory, as well as developing value stream maps and process models following the defined modeling standards. This will ensure that future improvement efforts will have maintained documentation to drive further improvement.

Transparency

One barrier to enabling the culture of continuous improvement is arming individuals with information about how processes work so they can create novel and innovative solutions. In many cases, detailed understanding is locked in a select few subject matter resources, which means any analysis efforts will be dependent on getting their valuable time. Having a transparent source of information that is accessible to everyone solves the issue of the limited subject matter experts and allows innovation to be democratized to everyone.

Ownership Model

Diffusion of responsibility [8] is a sociopsychological phenomenon wherein a person is unlikely to take responsibility when they assume that others in the environment are accountable. Essentially, if everyone is accountable, then no one is accountable. This hinders attempts to transform the culture toward a state where everyone actively drives improvements. A key strategy to avoid this situation is to clearly define accountability, set expectations and goals, and reward initiative and results on an individual level. Process Inventory plays a crucial role in this by defining ownership for each process.

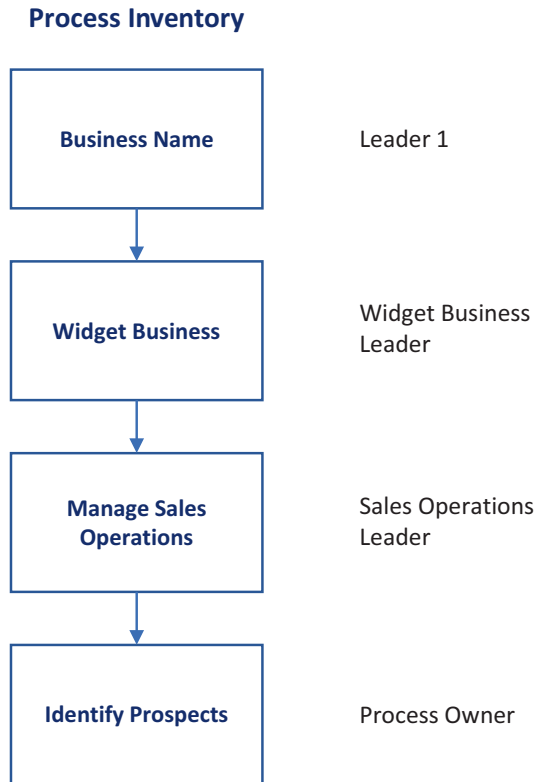


Figure 4-4. Process Ownership Model

Furthermore, the hierarchy of ownership, as illustrated in Figure 4-4, encourages everyone to contribute solutions that align with their scope of responsibilities and facilitates effective collaboration down the hierarchy. In this concept, lower-level process owners should be focused on ways to improve the process, while managers and leaders should be focused on higher-level strategies for success, such as whether is this process redundant and should be moved to a shared service team, should the process be outsourced to an external vendor, and do we have the right talent sourcing strategy.

Collaboration and communication along the hierarchy integrate perspectives at different levels and optimize processes for the organization.

Cross-Functional Coordination

Processes rely on various resources, including people, systems, data, controls, and vendors, all of which are managed and governed by distinct functional groups. By connecting all this information through the metadata library,

tracing ownership back to these groups becomes possible, enabling cross-functional validation before deploying solutions.

This integration is particularly useful as many processes are heavily dependent on technology for their performance. I've witnessed several Lean Six Sigma efforts conclude without actual improvements but with a list of system changes requiring prioritization for funding. Such clarity empowers business and IT leaders to collaborate to define an optimal IT environment aligned with the business's needs.

Leveraging Technology to Design Improvement

The advancement of technical capabilities can be harnessed to identify problems and devise solutions for process optimization. A prime example of such progress is the emergence of digital twins. A digital twin serves as a virtual representation of a physical object, system, or process, effectively capturing real-time data from its physical counterpart [9] to leverage artificial intelligence to conduct simulations, analyses, and predictions. This presents an exciting opportunity where technology aids in process analysis to pinpoint areas for improvement and define solutions.

Digital twins find their primary application in the manufacturing sector, where Internet of Things (IoT) sensors supply real-time data for their virtual counterparts to analyze. Across all industries, Process Inventory assumes the role of providing the virtual representation of all process areas, enabling technology to take on the role of an analyst. Further details on this concept are explored in Chapter 6.

Additional Operational Excellence Methods

While Lean Six Sigma is an important methodology for improving the performance and quality of processes, there are many other programs that contribute to achieving operational excellence. This highlights the crucial aspect that achieving operational excellence demands a comprehensive organizational effort where all functions are aligned toward the organization's objectives. Here are a few additional programs that hold significance, along with their ways that Process Inventory supports their success.

Enterprise Architecture

In today's digital age, the performance of processes is highly dependent on the systems and architecture they run on. Plus, a significant amount of operating costs is dedicated to maintaining the IT environment. Enterprise Architecture is responsible for defining the IT vision and governing the alignment of

technologies with processes and business strategies. By optimizing the IT infrastructure and integrating diverse systems, a well-run Enterprise Architecture function can promote agility and adaptability, driving continuous improvement and innovation across the organization.

Data Governance

The digital era has ushered in an unprecedented explosion of data. Data, if managed well, can be an asset that brings a competitive advantage. When managed poorly, it can lead to many operational issues such as inaccurate decision-making, poor process performance, low-quality customer service, high cost, and compliance issues. The data governance program is accountable for ensuring the availability, accuracy, and integrity of data. They establish data governance policies, define quality processes, design data architectures, unlock the power of data-driven insights, and more. A commitment to optimizing the data environment is a critical part of an operational excellence program.

Process Inventory and the details collected in process models, through data objects, provide data governance teams insights into which data each process uses and how it's leveraged. This information, especially when mapped to an enterprise data glossary or data model, can enable higher data quality by eliminating redundancies, inconsistencies, and inefficiencies in data storage, management, and usage. If process modeling is combined with system sequence diagrams, this can capture the end-to-end journey of data as it moves from source to destination, providing insights into transformations, bottlenecks, and compliance with data policies. Process Inventory provides a clear ownership model from which to coordinate data quality efforts.

Risk Management

Managing risk effectively holds significant importance for many organizations, particularly those operating in highly regulated industries. Establishing a program capable of proficiently identifying and controlling risks is paramount for informed executive decision-making, ensuring compliance with laws and regulations, mitigating financial risks, safeguarding business continuity, fostering customer trust, and upholding the organization's reputation.

The ways in which Process Inventory can support the effective management of risk are covered in [Chapter 7](#).

Operating Model Design

Leadership faces numerous considerations when structuring their organization, encompassing factors like insourcing vs. outsourcing, centralization vs. decentralization, geographical location, roles and responsibilities, and others. These decisions directly impact the organization's performance, which makes it a key consideration for an operational excellence program. An effectively designed operating model not only aligns business strategies with execution but also provides role clarity, eliminates redundancies, enhances customer service, and exhibits adaptability to changing circumstances.

Process Inventory offers leadership a transparent view of the processes performed and owned by each group and their alignment with the organization's performance objectives. When every process in the Process Inventory is correlated with a business capability model or a standard process model, it enhances visibility into potential process overlaps or redundancies. This heightened visibility serves as a catalyst for evaluating these processes, potentially leading to their rationalization through the creation of shared service functions and the standardization of similar processes. Such actions enable organizations to optimize their operating models to effectively align with the goals of operational excellence.

Supply Chain Management

Supply chain management plays a pivotal role in achieving operational excellence by optimizing the flow of goods, services, and information across the value chain. Effective supply chain management minimizes lead times, reduces costs, and enhances collaboration with suppliers and partners. It enables organizations to respond swiftly to market demands, reduce waste, and ensure consistent product quality.

Process Inventory supports supply chain management by linking the value chain in a value stream map and providing lower details to specific processes as previously discussed in this chapter.

Customer Experience

Providing superior customer experiences and engagement is a prominent goal in many business strategies. To deliver on that commitment, it requires a deep understanding of their customers, what motivates them, what pain points they experience, and how their needs evolve over time. This comprehensive understanding enables organizations to design products, services, processes, and experiences to address specific customer preferences and anticipate emerging trends. This has led many organizations to create dedicated customer experience functions that are focused on creating deep insights by studying their customers.

Chapter 2 covers the linkage between customer personas, customer journeys with processes in the Process Inventory, and the alignment of metadata; it enables traceability from the customer's perspective through each resource that supports those interactions. This enables precision in root cause analysis enabling the development of appropriate solutions to address issues or capture opportunities.

Process Inventory Powers Operational Excellence

This is just a small sampling as Process Inventory can power effectiveness in other programs such as vendor management, IT operations, issue management, and more. The key to optimizing how the organization operates and delivers on operational excellence goals is having a shared deep understanding of the organization's processes through Process Inventory. Process Inventory facilitates alignment to operational excellence goals, keeping each program focused on the organization's priorities. Process Inventory empowers the organization to streamline its operations, enhance quality, and achieve remarkable efficiency and adaptability in the challenging competitive landscape.

Key Takeaways

- **Operational excellence** is a state in which the organization is achieving and maintaining the highest level of performance aligned with its vision and goals. It requires an organization-wide purpose-driven culture of continuous improvement.
- **Lean Six Sigma** offers a cultural philosophy, rigorous and structured methodology, and a set of tools and techniques to aid in managing and improving quality and performance, as well as solving potentially complex problems to enhance processes.
- **Many more programs**, in addition to Lean Six Sigma, play a critical role in driving operational excellence success.
- **Process Inventory** powers effectiveness and facilitates alignment across all these programs toward operational excellence goals.

In the next chapter, we will delve into how the Process Inventory framework can transform the change process.

Transforming the Change Process

What You Will Learn

In this chapter, we will delve deep into the change process, a broad term encompassing any change in organizational capability, regardless of size and scope. You will gain an understanding of prominent change methodologies, including the strategy process, project portfolio management, Agile, Waterfall, and organizational change management.

We will explore how the Process Inventory framework can transform these methodologies by enabling alignment among all stakeholders and deliverables, ensuring that efforts remain focused on the strategic objectives of change. You will discover how it enhances the strategy development process, streamlines project portfolio management, provides clarity in scope and impact analysis, improves the handling of requirements and user stories, facilitates rigorous testing, and empowers effective change management for

people. Furthermore, you will find that project management becomes significantly more effective due to the transparency and traceability that the Process Inventory framework provides.

BBC Digital Media Initiative Transformation Failure

In 2008, the British Broadcasting Corporation (BBC) launched an ambitious project aimed at modernizing production operations and transforming how the organization manages data and provides content to audiences, all while reducing costs [1]. The business transformation program was named the Digital Media Initiative (DMI), which involved strategic investments in infrastructure, people, and production processes. A key aspect of the digital transformation was overhauling the BBC's data management system, with significant efforts dedicated to enabling staff collaboration on audio and video content and providing access to archived material.

From the outset, the program suffered numerous delays and rising costs. The BBC had awarded a contract, without running an open procurement process, to technology conglomerate Siemens to be their technology delivery partner. That contract was canceled a year later due to poor performance. The BBC brought the work in-house, but the struggles remained as they didn't have the technical capability to deliver the program.

In 2013, the BBC announced that DMI was shut down, and they would be writing off £98 million in unusable technology. These financial losses were a significant blow to the organization, drawing criticism and public scrutiny over the misuse of public funds. PWC performed a postmortem review [2] of how the program was managed and found numerous issues.

The program lacked adequate oversight and governance. It lacked an Executive Steering Committee, which could have challenged program management on progress toward their goals. The Finance Committee assumed some of that role, but their expertise primarily restricted them from challenging the financial costs and benefits. The program did not provide clear and transparent reporting on progress against the plan, cost to complete, and delivery of benefits. The risks and issues logs were not kept up-to-date and were missing several critical items, and the risks and issues captured focused on technology risks and issues rather than the ability of DMI to drive operational change to business practices. The program also lacked an effective assurance plan needed to audit program management practices and risks.

They lacked a solid approach to delivering the program. The BBC team didn't understand the methodology employed by Siemens. When they brought the work in-house, they lacked a structured approach with proper phases and

tollgate control to monitor delivery. The PWC report stated that they would have benefited from a proper Design Authority, which could have advised the program on the technical designs and the linkage to business outcomes.

Finally, it appears that the program was run predominantly from a technology point of view and lacked proper engagement from the business. The report found no evidence that the program had a business sponsor with an active role in the program. Their delivery roadmap didn't provide clarity as to when benefits to the business would be delivered, and they failed to engage business stakeholders to sign up for delivering those benefits.

Defining Change

Change is constant in organizations, but it becomes more prominent when an organization is adopting a digital agenda. There are several triggers for change, such as a shift in strategy or business model, response to competitive pressures, meeting customer expectations, adopting new technologies, regulatory changes, or any many other reasons. Implementing change can have impacts across one or more of the following: customers, products, services, risk and compliance, people, third-party vendors, employees, technology, data, or infrastructure.

Change can take on different forms based on its size and scope. There are large transformation efforts that redesign the organization's core capabilities to adapt to changing market conditions, capture new opportunities, and address significant challenges. Additionally, there are large mergers, acquisitions, or divestitures that require merging or splitting the resources of the organization within a compressed timeline. These endeavors also encompass significant changes and have wide-ranging impacts on the organization. These efforts typically span the enterprise, require multiple years of effort, and involve substantial funding, often in the range of tens or hundreds of millions.

On the other end of the spectrum, there are small projects focused on deploying small enhancements or fixing defects. Some changes may not involve technology at all and may instead entail organizational restructuring or cultural transformation.

Implementing change, especially for large transformation efforts, is one of the most complex endeavors an organization must undertake due to the following factors:

- **Coordination** across multiple stakeholder groups and practitioner teams, each with their own interests and perspectives, to deliver on the expected change.
- **Complexity** of the organization, its processes, and the systems means identifying the impacts and the precise changes that need to occur may take substantial effort.

- **Organizational culture** may be an impediment, especially for large change, as disruption to the status quo may be faced with resistance.
- **Interdependencies** can be difficult to understand and manage, missing key ones can result in delays or disruptions.
- **Risk**, when deploying change, can be heightened, or new risks can be introduced, requiring a deep understanding to implement and test appropriate mitigating controls.
- **Resource effort** can be significant, especially if the organization doesn't have the required skills in-house. Managing an external vendor adds another layer of complexity.
- **Duration** for large efforts may take years, which may result in fatigue especially if the change initiative requires long hours for key resources.

Change management is a systematic approach and set of methodologies, processes, and techniques employed to effectively navigate these challenges and deliver on the intended business objectives. It aims to address the complexities and uncertainties associated with change.

For the delivery of technology changes, there are several software development life cycle (SDLC) methodologies, which are processes for planning, creating, and deploying information systems. The two most prominent SDLC methodologies are Waterfall and Agile. While organizations may support both methodologies, the choice depends on the characteristics of the change effort and the specific project requirements. Additionally, two other disciplines play important roles in the change management process: organizational change management and project management.

Waterfall is characterized by a series of sequential phases, where each phase depends on the deliverables of the previous phase [4]. It follows a linear and structured flow and typically includes control tollgates for quality confirmation and approvals before progressing to the next phase. Waterfall is appropriate for complex system integration efforts, where the scope and requirements are stable, the project has a fixed timeline, and iteration is not required.

Agile is an iterative development process that emphasizes collaboration within cross-functional teams, including product owners, developers, testers, and DevOps members [5]. Agile is suitable when the solution has limited interdependencies, requirements are subject to change, and continuous product management input is needed. It is particularly effective when there is a focus on innovation and continuous improvement.

Organizational change management involves the planning, executing, and monitoring of activities that facilitate the adaptation of individuals and organizations to new ways of working and technologies. It entails recognizing and addressing the impacts on people, as well as understanding, preparing for, implementing, and sustaining changes within an organization. The ultimate objective is to maximize the benefits and minimize the negative impacts associated with those changes.

Project management is the practice of guiding a team to achieve all project goals within the given constraints. The discipline of project management is focused on planning, organizing, monitoring, and controlling programs or projects that are created to execute change.

Challenges in Executing Change

I mentioned in Chapter 1 that 70% of digital transformation failed to reach their goals. Those challenges extend to smaller efforts, and 9.9% of every dollar dedicated to change initiatives is wasted due to poor performance [8].

The top challenges in change initiatives, as defined by Forbes [9], are

- **Lack of Clarity and Execution Strategy:** The primary goal of projects is to solve a business problem. This requires a clear consensus among the entire group of stakeholders on the definition of the business problem and a robust execution strategy to deliver software that addresses the business objectives.
- **Inability to Reach Consensus on Priorities:** Project sponsors and project teams are not clearly aligned on the top priorities for the project.
- **Not Starting with the End Customer:** Change initiatives can begin with a great idea that is implemented, only to discover that the problem they solved wasn't the problem their customers faced. Doing the hard work of deeply understanding your customers, their desires, and what they're willing to pay for sets the ceiling on project performance and can help refocus a team when things derail.
- **Unclear Scope and Requirements:** A common reason for project failure is unclear requirements and a lack of detailed explanation of the vision for the initiative.

- **Working in Silos:** A major reason for failure is when teams embark on a journey to build something that is either not a business need or does not address the right problem, which is a symptom of misalignment between the business and technology.
- **Lack of Coordination and Detailed Planning:** Many projects fail due to a lack of good coordination and detailed planning. Teams need to implement a bottom-up planning process that identifies dependencies between deliverables and includes effort and duration estimates from critical resources.
- **Friction Caused by Undefined Roles:** Undefined or unclear role expectations create friction in project teams. Lack of clear accountability leads to quality issues in work products and the delivered change.

A Vision for Change Fueled by Process Inventory

In Chapter 1, I covered how the root cause of digital transformation failures is often a lack of strategic alignment. This same root cause underlies many challenges in the change process. Process Inventory addresses this by providing an anchor for all change efforts, regardless of size or type, to a comprehensive and accurate inventory of processes. It offers a consistent business-oriented lens through which to align vertically, from business strategy to the lowest level of implementation. Additionally, it enhances accountability and collaboration by aligning resources horizontally with the processes impacted by change. Process Inventory increases the effectiveness of organizational change management and project management by equipping these teams with more granular details on the impacts of change. Lastly, it establishes a sustained knowledge management structure that promotes continuous improvement and optimization.

Vertical alignment begins by aligning the outcomes of the strategic planning process with the process areas and customer touchpoints affected by the updated strategy. This enables the project portfolio team to assess priorities and estimate the size of change objectively, guiding the allocation of the change budget toward the highest priority business goals. Process Inventory provides a common language that serves as a business-oriented anchor for all deliverables in the change methodology, including scope, requirements, agile deliverables, design documents, testing, risk management impacts, and deployment and monitoring. This top-down traceability creates a linkage from leadership to lower-level resources executing the change, significantly minimizing any gaps in understanding.

Horizontal alignment is facilitated by identifying all impacted processes and the resources that support them. Each process in the Process Inventory framework is assigned an owner, making the accountability model clear. Furthermore, as each process is connected with the resources it leverages during execution, accountable resources within the various practitioner teams can be identified with precision. This not only facilitates effective planning by determining the scope for each resource but also allows for accurate effort estimation. Additionally, by detailing the interdependencies between processes, the Process Inventory framework enables identification of dependencies when delivering change.

By providing **organizational change management** teams with details on the impacted resources and how they will be affected, they can develop stronger strategies and plans to successfully guide the transition of individuals, teams, and organizations. This alignment ensures that all key deliverables, such as communications, training, and monitoring, are tailored to the specific needs of those impacted, thereby increasing the chances of a successful transition. Furthermore, this alignment fosters better relatability between individuals and teams when leadership communicates their vision for the future.

The effectiveness of **project management** is enhanced by the traceability and transparency provided by the Process Inventory framework. Project plans and program roadmaps are supported by increased detail on impacts and the involved resources. Stakeholder engagement becomes clearer through the accountability that Process Inventory provides. Status reports offer enhanced clarity to leadership as they can communicate progress in business terms, such as the progress of processes or process areas toward the project goals. Figure 5-1 illustrates how traceability is accomplished throughout the life cycle of change delivery. As each output, from strategy through postproduction monitoring, is aligned to the process, it provides a business-oriented connection that stakeholders can understand when validating whether project objectives are being met.



Figure 5-1. Change Process Traceability

Most deliverables in change initiatives are transitory, meaning they are created solely for a specific effort, are archived when that effort is complete, and are not typically leveraged in future initiatives. This is often due to the use of different organizing frameworks for each effort, making it challenging to correlate past deliverables to current project needs. Process Inventory, by providing a consistent framework, allows deliverables and the knowledge

within them to be leveraged by subsequent efforts. This promotes **continuous improvement and optimization** of processes and documentation by capitalizing on the collective wisdom and lessons learned from previous initiatives. This means that initiatives will no longer begin with a “blank sheet of paper.” Instead, teams can direct their energies toward improvements rather than spending significant effort on understanding how things work.

The Change Initiative Process

Figure 5-2 represents the change initiative process, which goes from developing a strategy through deployment and monitoring of changes. I will leverage this through the remainder of the chapter to detail various phases and to demonstrate how Process Inventory supports much greater effectiveness of the change process.

It’s important to note that not all change initiatives will involve changes to processes as some may be focused on infrastructure or cosmetic user interface changes. However, understanding the impacts of these changes on processes is still important.

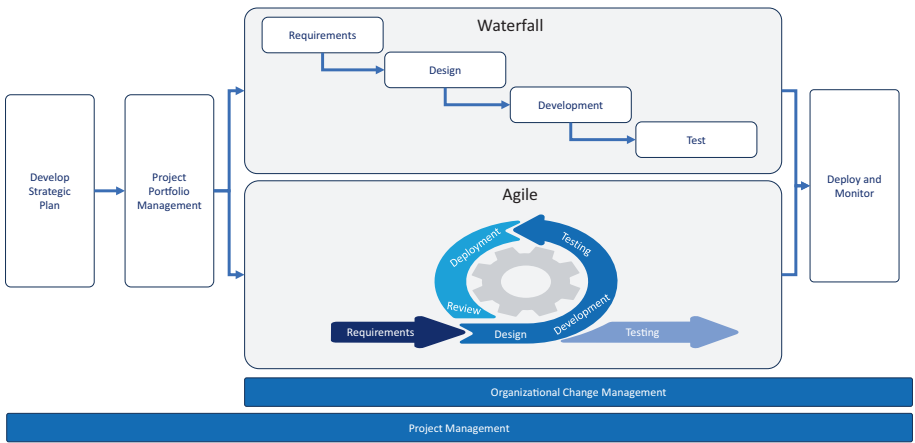


Figure 5-2. The Change Initiative Process

Develop Strategic Plans

Strategic planning is the organizational process of defining its strategy or direction and making decisions on allocating resources to achieve strategic goals [10]. By envisioning the desired future state and aligning strategies and actions accordingly, strategic planning shapes the plans and decisions of the organization. In many organizations, this is performed yearly, and it provides a

roadmap that guides investments, goal setting, and performance evaluation, ensuring that the organization moves in a purposeful and coordinated manner toward its long-term objectives.

In many organizations, the primary goal of strategic planning is to identify the organization’s direction and priorities for the next three to five years. Figure 5-3 illustrates a typical strategic planning process.

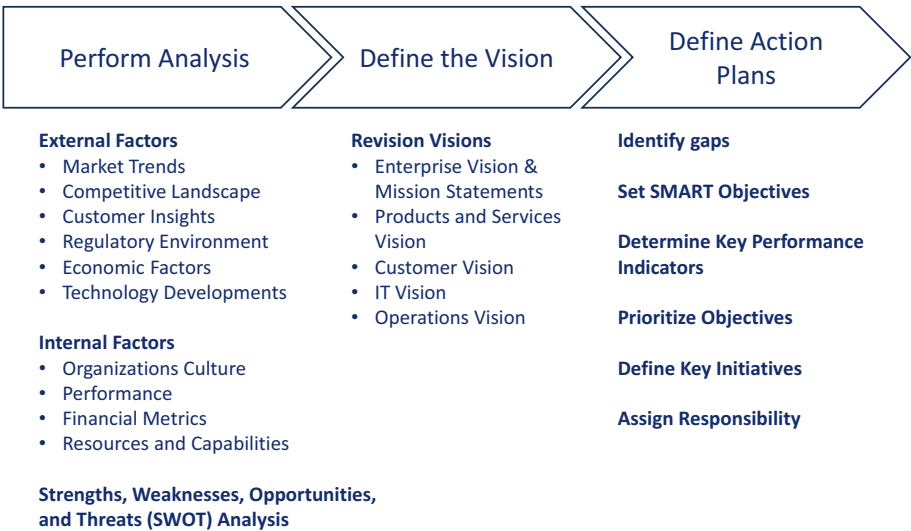


Figure 5-3. Strategic Planning Process

Perform Analysis

During this phase, various external inputs are gathered, including market trends, the competitive landscape, customer insights, changes in the regulatory environment, economic factors, technology developments, and other relevant external information. If the organization has a customer experience (CX) team that has been creating customer personas and customer journeys, this presents an excellent opportunity to incorporate their valuable insights.

In addition, internal inputs are gathered, including operational performance, financial performance, organizational culture, resources, and more. Process Inventory can support this process in two ways: Firstly, strategists can use it to identify each process area or capability as strategically important or foundational. Secondly, if the organization has aligned process performance metrics to the Process Inventory framework, this provides an opportunity to leverage this insight to identify important processes or capabilities that are underperforming.

These two inputs provide the basis for a SWOT analysis, which is a brainstorming tool to identify the organization's strengths, weaknesses, opportunities, and threats. This comprehensive analysis aids leaders in understanding the strategic changes required for the organization.

Define the Vision

A vision and mission statement represents the organization's purpose and value proposition, outlining its long-term aspirations and goals. Well-crafted vision and mission statements inspire and motivate employees, stakeholders, and customers by providing a clear sense of direction and purpose. While vision and mission statements are typically stable, they can be revisited and refined through this process. This can lead to changes in future state visions for other key aspects of the organization, such as products, services, customers, technology, and operations.

At this point, a vision for the future state of processes should be developed. Not all processes have the same importance as some are critical for customer interactions, profitability, or other reasons. Identifying which processes, through the Process Inventory framework, are of high strategic importance will help allocate initiative investment in subsequent steps.

Defining Action Plans

In this step, the organization's vision is compared to current capabilities to identify gaps that require investment. These gaps are converted into SMART objectives, which specify the objectives as specific, measurable, achievable, realistic, and timely. These objectives will be logically grouped based on how they will be implemented into change initiatives. Examples of these initiatives include mergers, acquisitions, divestitures, outsourcing process areas, new revenue streams, technology transformation, and more.

These initiatives will be mapped to the organization or organizations that are responsible for their implementation and will be impacted by them. By aligning them to the Process Inventory, this mapping can be done with precision and can lead to the identification of resources across all practitioner teams that will be responsible.

The output of the strategic planning process is typically reviewed and approved by the senior leaders of the organization and most likely the board.

Portfolio Planning Process

Portfolio planning is the process of evaluating initiatives to determine which, out of a finite amount of funding, provide maximum value to the business and are thus allocated funds. This process also enforces rigor to ensure that each project requesting funding creates a business case with the appropriate level of diligence in identifying scope, estimating costs and benefits, identifying risks, and addressing other factors that support the successful delivery of the project. Figure 5-4 illustrates the steps in the portfolio planning process.

US LIFE AND ANNUITY INSURANCE PROVIDER

A global insurance provider strategically determined that their US life and annuity business did not align with their future vision, leading them to decide to spin off this business segment.

The newly formed company embarked on a substantial transformation program to separate their people, processes, and technology from the global parent organization. They sought consulting assistance in defining the program, establishing priorities, and developing a multiyear roadmap.

Prior to initiating the program, the company established key principles that would guide their design decisions for the new organization. Their value proposition centered around being a low-cost insurance provider compared to their competitors. Consequently, they aimed to avoid inheriting any legacy systems, opting instead to build their infrastructure from scratch. Their vision included a cloud-based environment, leveraging vendor platforms, and outsourcing nonstrategic business processes. Essentially, they operated as a startup with substantial total assets and existing insurance policies.

When they approached us for a proposal, they expressed a desire to adopt a “City Plan” approach, where the program design would align with their functional areas. I was brought in to lead the proposal writing process, and upon winning the project, I assumed responsibility for building the engagement team.

Our initial step involved defining their project portfolio. We created a map of their functional areas, akin to a business capability model. The company’s leadership then used this map to categorize each area as either “best in class,” indicating a strategic differentiator requiring increased investment, or “best in cost,” representing areas that needed to be efficient but didn’t necessitate extensive investment.

Working closely with each functional area, we conducted sessions to define their inventory of processes and establish high-level requirements for each process. This information informed the development of request for proposal (RFP) requirements for

vendors to support the initiative, as well as helped shape the portfolio of initiatives. We crafted a charter for each initiative, outlining the scope, effort and financial estimates, timelines, and key risks associated with each.

The culmination of these efforts resulted in an integrated roadmap for the program, taking a process-centric view. This roadmap served as a guide throughout the execution of the program, ultimately leading to its successful completion.

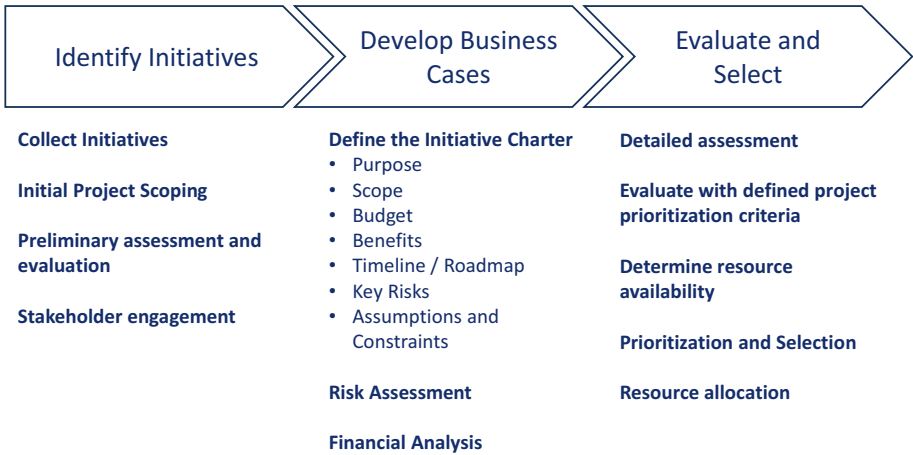


Figure 5-4. Project Portfolio Management

Identify Initiative

In this phase of the project portfolio process, all proposed initiatives are brought to light. While the strategic planning process contributes to some of these initiatives, it's important to note that each business unit or functional area may have its own proposed initiatives that align with its specific goals, which may include the continuation of funding for initiatives that have started in previous cycles.

During this phase, the project team engages with stakeholders from various areas and conducts initial scoping and evaluations based on the projected budget. The aim is to assess the feasibility and alignment of each proposal with the organization's strategic objectives. Through this process, some proposals are down selected and excluded, while others progress to the next step in the evaluation process.

Develop Business Cases

Initiatives that progress to this phase are required to develop a detailed business case, which is captured through an initiative charter. Initiative charters are applicable to both programs and projects. Programs generally encompass larger-scale initiatives that consist of multiple coordinated projects aimed at delivering significant value for the business. On the other hand, projects are scoped to deliver more focused changes, targeting specific objectives. The initiative charter serves as a formal document that outlines the goals, objectives, scope, resources, timelines, and expected outcomes of the proposed initiatives. It provides a comprehensive overview and foundation for effective planning and execution during the subsequent phases of the project portfolio process.

Figure 5-5 illustrates how Process Inventory can provide alignment for a change initiative.

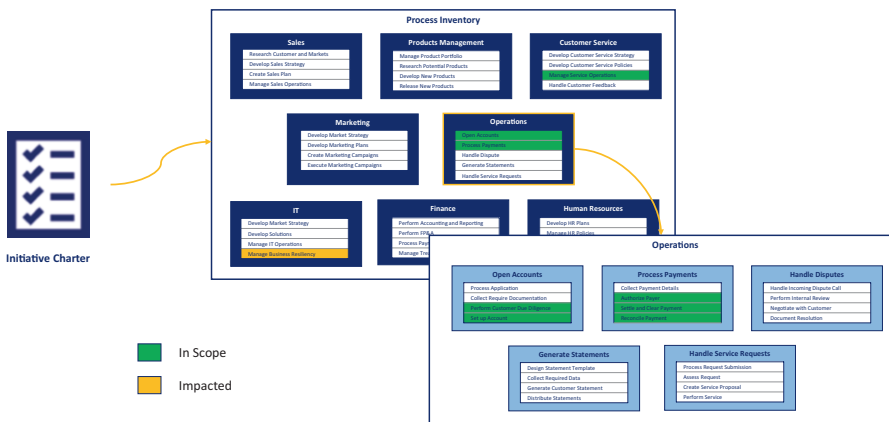


Figure 5-5. Define Initiative Scope

By leveraging Process Inventory, organizations can **define scope** with precision for programs or processes. This approach ensures that the boundaries of scope are clearly established using a business-oriented lens, enabling all stakeholders to understand and align with it. In addition, the preceding diagram distinguishes between processes that will change as part of the initiative and those that will be impacted. Impacted processes require regression testing to ensure that no errors are introduced due to changes in underlying resources.

Furthermore, Process Inventory enables programs to conduct detailed **impact analysis** by identifying process owners and understanding the resources associated with each process. This information facilitates more accurate effort, budget, and timeline estimation as it provides a comprehensive

view of all required changes. Additionally, aligning performance metrics with Process Inventory allows for easier calculation of benefits derived from the initiative.

Risk assessments are completed with greater accuracy through the detailed information obtained via Process Inventory and its connection with risks and controls documented in the GRC (governance, risk, and compliance) repository. This comprehensive approach ensures that risks are properly identified, assessed, and managed, leveraging a thorough understanding of the underlying processes and associated controls. It highlights the value of leveraging Process Inventory to strengthen risk management practices and enhance the overall effectiveness of the organization’s GRC framework.

Evaluation and Selection

Alignment of scope with business processes and improved precision in assessing impacts, dependencies, and cost estimates enable portfolio teams to make more objective assessments when selecting projects for funding. By prioritizing initiatives that align with the strategic importance of process areas, portfolio teams can enhance the efficiency of change funding, ensuring that necessary changes are effectively implemented.

Requirements

According to the Project Management Institute’s (PMI) 2021 Pulse of the Profession report [11], 40% of project failures can primarily be attributed to poor requirements. The cost of addressing requirement defects in software rises exponentially as it progresses through the life cycle up to production deployment. In one estimate [12], requirements errors cost US businesses more than \$30 billion per year.

MERGER OF REGIONAL BANKS

Two regional banks of approximately equal size had announced a merger and had set up a post-merger integration program to reach Client Day One, the day when the combined company operates as a single entity and serves its customer base under the new brand.

As part of their program, they recognized that future state process design was critical to a successful transition. They sought consulting help from a partner with expertise in large-scale process design.

I led the team that developed the request for proposal (RFP) response and ultimately in delivering the work. Our scope was to create future state designs for eight business units. Prior to our onboarding, the bank had already made decisions on the platforms and personnel they would leverage in their future state processes. Our engagement began with creating process inventories for each business unit and then process maps for high-priority processes.

This turned out to be a valuable input, providing clarity on the integration impacts and enabling better requirements for the system, personnel changes, and risk management.

Waterfall Business Requirements

Requirements are the needs of the system, typically expressed by business stakeholders, end users, and other subject matter experts. They encompass the functions, features, and behavior of the software system being delivered. Requirements provide traceability to the scope outlined in the charter, as well as to many deliverables in subsequent phases. They usually consist of high-level requirements that are then further detailed through low-level requirements. Final approval of requirements often rests with the business owner.

Requirements are completed upfront and remain relatively stable throughout the project life cycle. They form the basis for establishing the timeline and effort estimation. Most projects have a change control process in place, where proposed changes to requirements undergo evaluation for their impact on schedule and costs before being approved.

Agile User Stories

Agile is a different approach where requirements are not fully defined upfront but iterated on through the collaborative process of design, build, and test cycles called sprints in cross-functional scrum or feature teams.

User stories are short descriptions of functionality expressed from a user's perspective. They capture the "who," "what," and "why" of system requirements and are often documented in the format: "As a [type of user], I want [an action] so that [a benefit/value]." User stories are accompanied by acceptance criteria, which define the specific conditions that must be met for a user story to be considered complete. Agile teams commonly use story points to estimate the relative effort or complexity of user stories.

User stories are small, independent, and testable units of functionality that can be implemented within a sprint, which is typically two weeks but can vary in length. They are organized into epics, which are large customer-centric user stories encapsulating broad initiatives or features too large to develop in a single iteration. Epics can be part of themes that represent high-level business objectives or areas of focus. Themes and epics provide broad categories aligning with strategic goals, aiding in prioritization and work organization.

The product owner, working in the scrum team, takes the lead in developing user stories and adding them to a product backlog, which serves as the team's to-do list. Agile teams work in sprints to develop the product. Before each sprint, user stories are selected from the product backlog during the sprint planning process. At the end of the sprint, a demo is conducted to showcase completed features and gather feedback from business stakeholders.

User stories in the backlog undergo continual refinement throughout the iterative process as the scrum team and business stakeholders gain a better understanding of their requirements and preferences. Backlog grooming or refinement is an ongoing process where the product owner and development team regularly review and prioritize the product backlog, ensuring that user stories are up-to-date, well-defined, and aligned with the evolving needs of the project and stakeholders.

Top Challenges in Creating High-Quality Requirements

Regardless of the delivery methodology being followed, high-quality requirements or user stories are critical to successfully delivering change. Here are the top reasons for poor requirements:

- **Inadequate Stakeholder Involvement:** Various stakeholders, such as product managers, business owners, risk managers, subject matter experts, customers, users, and regulators, may have an interest in the project. While it may not be feasible to gather inputs from all stakeholders, it is crucial to identify the set of stakeholders that can cover all necessary perspectives.
- **Insufficient Domain Knowledge:** Insufficient or incomplete understanding of the business domain can lead to requirements that fail to address the business problem, are unrealistic, or do not align with regulatory needs. In large change initiatives, a lack of understanding of upstream and downstream connections can result in defects in the end-to-end process.

- **Ambiguous Language:** Requirements that are not clearly articulated can lead to misunderstandings and misinterpretations by implementation teams. However, this challenge is less pronounced in Agile methodologies due to the collaborative nature of the scrum team.
- **Insufficient Validation and Verification:** Failure to perform quality reviews to validate that the requirements meet stakeholder expectations and verify the accuracy and completeness of requirements during their creation can lead to undetected quality issues and conflicting requirements.
- **Lack of Requirements Documentation and Traceability:** Properly documenting requirements in a clear and accessible manner is important for other stakeholders to understand. Traceability involves establishing linkages to concepts in the project life cycle, such as scope, high-level requirements, and testing scripts. Failure to do so can result in confusion and gaps in requirements.
- **Poorly Defined Change Control Process:** In Waterfall projects with long durations, business needs or stakeholders' understanding of their needs may evolve, leading to requests for changes to requirements that have been locked down. Without a rigorous change control process that identifies the impacts of change requests on the timeline and budget, there is a risk of scope creep and the project becoming larger or spinning out of control due to frequent changes.

Some of these challenges may not apply to a single Agile team since they are built around what Amazon labeled the “Two Pizza Team,” which means teams that are small enough to be fed with two pizzas. If the project or program is larger and requires coordination of multiple Agile scrum teams to deliver a larger set of functionalities, then these challenges do become relevant.

Benefits of Documenting Requirements After Processes

Process Inventory can address many of the challenges in business requirements by providing alignment as mentioned earlier. Figure 5-6 illustrates the correlation between the level of detail in business requirements for Waterfall and user stories for Agile and the depth of the Process Inventory. This enables

leadership to express their requirements from their perspective and have project teams further detail them at lower levels.

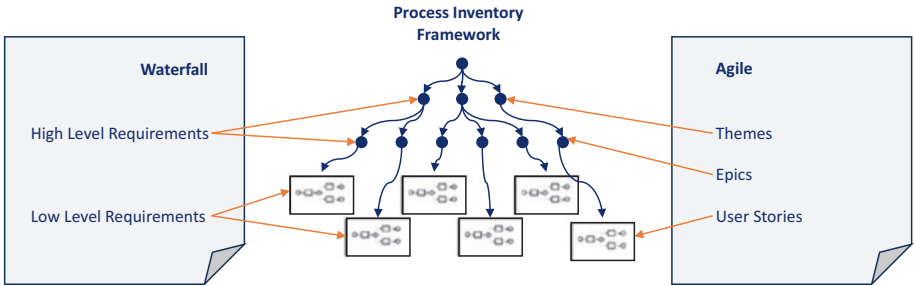


Figure 5-6. Alignment of Requirements to Process

The initial step in this model involves updating the Process Inventory to reflect any changes in the organization’s processes. Processes may be added in response to new products, features, revenue streams, or acquisitions, while processes may be removed due to divestitures, outsourcing, product retirements, or process optimization. It is important to note that this list is not exhaustive as other factors can contribute to changes in the Process Inventory.

This step is crucial as it allows stakeholders to visualize the impact of the change on the organization’s processes. It provides a comprehensive understanding of the business domain and scope, facilitating stronger collaboration among stakeholders. It also establishes a consistent structure that enables effective leveraging of existing knowledge.

Creating future state process models as the next step, prior to documenting requirements, enables product managers and process owners to align their understanding of strategic and project objectives with changes to how processes will be executed. It also ensures alignment from customer interactions through product design to underlying systems and behaviors. This step enables the identification of gaps and interdependencies, such as upstream and downstream connections, impacts on customers and employees, risks, and the use of third-party vendors and resources. By addressing these aspects, the resulting requirements comprehensively cover the necessary aspects of the project.

Furthermore, it serves as the foundation for traceability, a crucial aspect that can significantly enhance development projects. Research shows that traceability can accelerate project speed by 24% and improve project accuracy by 50% [13]. Traceability refers to the ability to track a requirement’s life cycle across project deliverables, including software code, both forward and backward. The Process Inventory framework facilitates traceability from

strategic objectives all the way to code deployment in production, as illustrated in Figure 5-1. By establishing traceability, organizations can effectively monitor and trace the origin of defects, enabling faster detection, analysis, and resolution of issues throughout the development process. As a result, there is a reduction in software defects and an improvement in the overall quality of software maintenance.

Taking a top-down approach by starting with future state process models yields higher quality and more complete requirements. Stronger validation and verification are achieved by enabling requirement teams to trace each requirement to its support of the future vision of processes. This approach also leads to process optimization by identifying and addressing redundancies, inefficiencies, and poor resource utilization during the implementation of change.

By building requirements from a complete set of impacted processes, this facilitates accountability by involving the process owner and business stakeholders in attesting to their completeness and accuracy. Communication with senior stakeholders becomes more effective as they can gain a rolled-up view to ensure that the requirements reflect their expectations.

Risk and Compliance Requirements

Managing risk and complying with regulatory obligations are critical for many industries and organizations. The change process, particularly for large transformational initiatives, can pose significant risks if essential controls are overlooked during deployment.

That's why it's crucial to integrate risk managers, including the first line of defense (business units or functions) and the second line of defense (risk oversight), into the requirements process. Implementing the solution outlined in Chapter 7, which involves aligning risks, controls, and regulatory obligations in the GRC (governance, risk, and compliance) repository with the processes identified in the Process Inventory, would enhance the effectiveness of the risk management teams in documenting risk requirements. They would be able to identify whether changes to existing processes introduce new risks or regulatory obligations and determine the need for new controls or updates to existing controls. In the case of implementing new processes, it allows them to review the processes defined in the future state Process Inventory and identify the additional controls required.

Risk managers are responsible for ensuring that requirements adequately cover the necessary controls, thereby collaborating more effectively with the project team to reduce risk during the solution's deployment.

Furthermore, involving the risk management team early in the process offers the valuable benefit of shaping the implementation of controls for enhanced effectiveness. Many organizations strive to automate controls, transitioning from manual and detective measures (after the risk is incurred) to proactive and preventative measures (before the risk is incurred). By including the risk management team from the outset, their expertise can contribute to optimizing control mechanisms and establishing a more efficient and robust risk management framework.

Testing

Testing plays a crucial role in the software development life cycle, serving as a vital step to prevent defects or quality issues from being released into production and impacting customers or damaging an organization’s reputation.

There are two primary types of testing: functional testing and technical testing. Functional testing ensures that the software delivers the intended functionality as outlined in the requirements, while technical testing focuses on assessing the software’s robustness and its adherence to necessary technical standards. This type of testing encompasses evaluating performance, security, scalability, and other technical aspects to ensure the software’s overall soundness.

Within a project context, functional testing holds significant importance as it validates whether the project’s original objectives and scope, as expressed through the requirements, have been successfully met. Figure 5-7 illustrates how this validation process aligns with the V-model framework.

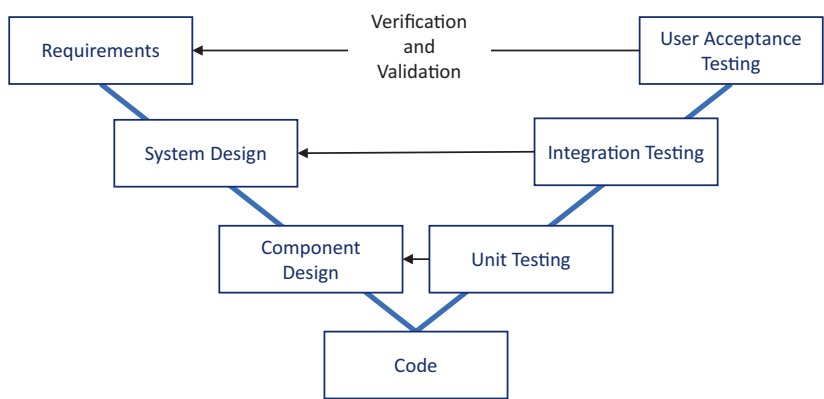


Figure 5-7. V-Model

As the model demonstrates on the left side, progressing downward, there are different levels of granularity involved in defining the solution, starting from requirements and extending to software code. On the right side, different

levels of testing are depicted, each serving a specific purpose in terms of verifying and validating artifacts. The highest level of functional testing, user acceptance testing, plays a critical role in validating that the requirements have been satisfactorily met and that the solution effectively meets the needs and expectations of end users. This level of testing serves as a pivotal assurance to stakeholders that the software is ready for deployment and aligns with the defined business objectives.

While the V-model has traditionally been associated with Waterfall methodologies, adaptations have been made to accommodate Agile environments. The fundamental principle remains intact: the delivered software code must align with the Agile requirement artifacts, including themes, epics, and user stories. However, the key distinction lies in the iterative nature of Agile practices, where testing activities are integrated throughout the development process. This approach allows for continuous validation at each iteration or sprint, ensuring that the evolving software aligns with the desired objectives and meets the evolving needs of the stakeholders.

Benefits of Aligning Test Script to Process

If the alignment of requirements to processes, as defined through the Process Inventory framework, is adopted, it logically makes sense to align scripts in user acceptance testing with processes as well.

There are several benefits to this approach. The primary advantage is that it provides alignment across all stakeholders, such as the product manager, product owner, and risk manager, ensuring that requirements have been successfully met through testing. It also enables end-to-end testing from the customer's or end-user's perspective, validating that the system fulfills their needs.

Since the Process Inventory framework will outlive the project's duration, it offers an excellent structure for building a library of regression testing scripts. Regression testing involves executing test scripts at a later point in time to ensure that the systems continue to function as expected, even after changes such as defect fixes, software upgrades, or infrastructure modifications.

Furthermore, this consistent structure enables the development of an automated testing library, which reduces the time and cost associated with testing while improving accuracy and providing valuable insights.

Organizational Change Management

Organizational change management (OCM) is a discipline that focuses on effectively managing change to facilitate successful transitions of individuals and teams from the current state to the desired future state. While OCM

encompasses technology-driven changes, such as those driven by Agile and Waterfall projects, it also extends to non-technology-driven changes, including strategy, organizational structure, culture, processes, and operations.

The primary objective of OCM is to maximize adoption, minimize resistance, and ensure the overall success of the change effort by employing structured methodologies, tools, and techniques. Central to OCM is the recognition of the importance of addressing the human aspects of change and gaining organizational buy-in to embrace new ways of working.

The adoption of digital technologies, especially artificial intelligence, will have a substantial impact on workforces. These technologies will act as productivity multipliers by reducing or eliminating repetitive tasks, affecting almost all roles and potentially making some obsolete. This level of disruption may induce fear and resistance among certain parts of the workforce who are inclined to defend the status quo. Resistance can be significant, as evidenced by a McKinsey study that found an average loss of 20% of a transformation's value after program completion [14], highlighting the importance of maintaining a long-term focus on OCM execution.

SOUTHWEST AIRLINES CULTURE FUELS ITS BUSINESS PERFORMANCE

Southwest Airlines' remarkable success can be attributed to its strong organizational culture, which has propelled the company to be profitable for more than 45 consecutive years and receive recognition as one of Fortune's Most Admired Companies for over 23 consecutive years [15].

Unlike many organizations that prioritize customer satisfaction, Southwest places a higher emphasis on the well-being of its employees. The company's hiring process revolves around selecting individuals who align with the company culture, focusing on attitude rather than solely on skills.

Furthermore, Southwest provides its employees with a unique and empowering work environment, fostering a sense of autonomy and offering ample opportunities for personal growth. This approach has led to a low employee turnover rate, with internal promotions being the norm within the company. Southwest's commitment to supporting and retaining top talent is well-known as the company openly states that employees will never be penalized for using good judgment and common sense in favor of providing exceptional customer experiences.

Former CEO and current chairman of Southwest, Gary Kelly, emphasizes the importance of the company's core values, known as the "Southwest Way," which include a warrior spirit, a servant's heart, and a fun-loving attitude. These values have been ingrained in the organization's culture for 36 years and continue to shape the

company’s operations and employee behavior. Southwest’s dedication to maintaining its culture is evident through initiatives like the “Corporate Culture Committee,” which promotes and sustains the “Southwest Way” by recognizing employees who deliver outstanding service.

Southwest Airlines’ success story showcases how a strong organizational culture focused on employee well-being, customer service, and core values can drive business performance, foster employee engagement, and create a positive and enduring company reputation.

It’s imperative that organizations develop a comprehensive workforce strategy to mitigate the downsides of digital disruption. However, this extends beyond digital as having a strong workforce strategy and a motivated, dedicated workforce is critical to business performance and long-term success. A well-defined workforce strategy ensures that an organization has the right people with the necessary skills and capabilities to meet its current and future needs. As illustrated by the Southwest Airlines example, by aligning the workforce strategy with business objectives, organizations can optimize performance, drive sustainable growth, and effectively navigate the challenges posed by digital disruption and other transformational changes.

Process Inventory plays a crucial role in achieving alignment and driving success in the organizational change management (OCM) process. Figure 5-8 provides a visual representation of this process, which can be executed independently for non-technology-driven changes or run parallel to technology-driven change.

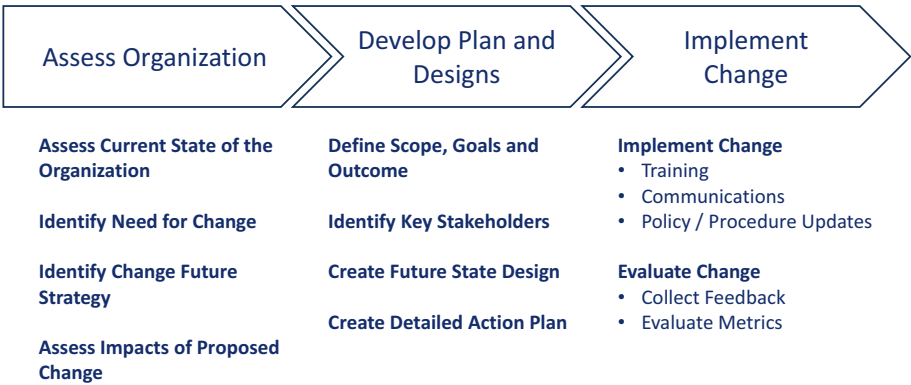


Figure 5-8. Organizational Change Management Process

Assess Organization

Process Inventory provides leaders with enhanced visibility into the organization's processes, allowing them to understand the current state and align their future state aspirations with the business objectives. This includes assessing the impacts of various types of change, including technology-driven change, to determine how teams and individual roles may be affected. This clarity enables leaders to proactively identify potential risks or challenges before launching the program.

Develop Plans and Designs

With its organizationally aligned structure, Process Inventory facilitates the identification and engagement of stakeholders. Detailed action plans can be created with precision by understanding all the artifacts, such as processes, training, policies, and procedures, that need to be updated or created to support the change. Plans can be refined with clear steps, milestones, and measures to determine success.

Implement Change

During the implementation phase, communications regarding the changes are conducted, procedure documentation is updated, and training is rolled out to individuals and teams. Stakeholders, adopting ownership and influencer roles, play a crucial part in championing the change to their managers and teams. Additionally, OCM teams remain engaged throughout the process, collecting feedback, evaluating metrics, and adjusting implementation strategies if necessary.

Project Management

The discipline of project management is critical to the success of implementing any change, and it is imperative for delivering complex transformation programs such as digital transformations. According to the Project Management Institute, projects that employ proven project management practices are 2.5 times more likely to succeed and waste 13 times less money [16].

Project and program managers play a pivotal role in guiding the delivery of change from initiation to production, utilizing proven techniques and tools to ensure accountability among team members and provide transparent communication to senior stakeholders. They are responsible for documenting and maintaining project scope, creating and managing delivery plans, providing comprehensive status reporting and effective communication to leadership, planning and allocating resources and finances, managing delivery risks, and

conducting ongoing evaluations for continuous improvement. Their expertise and diligent management contribute significantly to the successful execution of projects, helping organizations achieve their desired outcomes with confidence.

Process Inventory Strengthen Project Management

Process Inventory bridges the gap between senior sponsors and execution teams, bringing clarity and alignment between those funding the project and those responsible for delivering on its objectives. As discussed earlier in this chapter, this process-driven approach enables clarity in defining scope, detailing impacts and plans, documenting requirements, aligning testing, and managing organizational change. In this section, we will explore how Process Inventory supports project managers in monitoring and controlling project delivery and facilitating transparent progress reporting to senior stakeholders.

Project Planning and Work Estimation

Delivering successful change requires effectively managing the constraints of scope, cost, and time. To achieve this, it is crucial to establish work packages that align with the scope defined in the Process Inventory. By framing the project scope in a business-oriented context understood by all stakeholders, work packages play a vital role in accurately estimating the cost and duration of implementing desired changes.

To illustrate this concept, let's consider a project with a scope consisting of five processes that require updates. Each process becomes a distinct work package, encompassing its specific set of requirements, design documents, software components, testing procedures, and other deliverables. This logical breakdown of the project into work packages provides clarity and assigns responsibility at the process level, ensuring accountability and ownership.

By breaking down deliverables into work packages, project managers can estimate the cost and duration associated with each package more precisely. Estimating each individual deliverable within a work package allows for better aggregation of the overall project cost and duration. This level of granularity enhances project planning and resource allocation, enabling effective constraint management.

Moreover, work packages contribute to aligning stakeholder expectations and understanding by providing a clear breakdown of responsibilities for each deliverable. Team members can easily identify their roles within specific work packages, promoting accountability and fostering collaboration throughout the project life cycle. This focused ownership facilitates efficient communication, streamlined decision-making, and effective coordination, ultimately driving successful project outcomes.

It is important to acknowledge that this approach has limitations, particularly when the change does not involve process changes, such as infrastructure-oriented changes or cosmetic updates to user interfaces. Additionally, while this level of granularity and structure may not be necessary for small agile projects, it becomes increasingly important when multiple scrum Agile teams need to coordinate and align their efforts to deliver transformative programs.

Monitoring Progress

By breaking down the work efforts associated with process-aligned work packages, this leads to the project management technique of measuring progress through earned value management (EVM). EVM provides a holistic view of the project’s health by calculating the value of the work performed against the planned value, as determined by the initial estimate, and comparing it with the actual cost. This enables project managers to calculate the estimate-at-complete, providing them with a real-time understanding of the impacts on the planned budget and timeline.

The rigor of this approach enables project managers to calculate the cost and timeline implications of proposed scope changes through the change control process. This provides decision-makers with accurate information to determine the trade-offs associated with accepting changes to the project’s scope.

Process Aligned Status Reporting

The Process Inventory approach leads to more transparent reporting of status for leadership. By framing progress through processes or process areas, which they can easily relate to, leadership gains a view of how progress aligns with the ultimate business objectives. This alignment provides clarity in terms of accountability for addressing delivery issues or risks, enabling leadership to make informed decisions and take necessary actions. Figure 5-9 showcases a process-aligned status dashboard, providing an illustrative view of how Process Inventory can be effectively presented.

Scope	Schedule	Budget	Requirements	Design	Development	Test
Process 1			100%	100%	25%	-
Process 2			75%	-	-	-
Process 3			100%	25%	-	-

Figure 5-9. Illustrative Process Aligned Status Dashboard

Tollgate to Production

An important mechanism that project managers use to control the delivery of projects is tollgates. Tollgates are checkpoints that occur typically after major phases, where project management and other critical stakeholders can review the quality of deliverables, assess delivery risks, review issues, and determine if the project has met its milestones and is on track to meet its business objectives. This gives senior sponsors and stakeholders the ability to determine if work can proceed to the next phase or if major issues need to be addressed before moving on.

The final tollgate before a change is implemented is the Go/No Go decision, where senior stakeholders can determine if the project will meet its business objectives and, most importantly, ensure that implementation won't incur unnecessary risk because a critical component isn't ready. For complex transformation programs, there may be many moving parts that must be coordinated to make this happen. New technology must not only be tested and defect-free, but the deployment plan, which may require complex sequencing and migration of data, must also be in place. Employees need to be properly trained and ready to operate in the new environment. Communications to customers will have to be completed to ensure a seamless experience. Risks must be understood, and the proper controls must be in place. There are many other items to be coordinated, and the Go/No Go tollgate gives leadership visibility that everything is ready, and the implementation can proceed.

Process Inventory promotes stronger tollgates through the alignment of deliverables, outcomes, and stakeholders with the impacted processes. This business-oriented perspective ensures that updates are presented in a way that stakeholders can understand. The framework's emphasis on accountability enables each stakeholder, including product managers, process owners, risk managers, and technical leaders, to verify that the change has met their expectations. This instills confidence in senior stakeholders, particularly during the final Go/No Go decision as they receive accurate and comprehensive information prior to making their decision. This structured approach enhances the success of change deployment.

Key Takeaways

- **Change process** is a broad term encompassing any change in organizational capability, regardless of size and scope.
- **Prominent change methodologies** include the strategy process, project portfolio management, Agile, Waterfall, organizational change management, and project management.

- **Process Inventory framework** can transform these methodologies by enabling alignment among all stakeholders and deliverables, ensuring that efforts remain focused on the strategic objectives of change.
- **Numerous benefits of this approach** include enhancing the strategy development process, streamlining project portfolio management, providing clarity in scope and impact analysis, improving the handling of requirements and user stories, facilitating rigorous testing, and empowering effective change management for people.
- **Project management** becomes more effective due to the transparency and traceability that the Process Inventory framework provides.

In the next chapter, we will delve into how the Process Inventory framework can enable stronger management and design of the technology environment.

The Technology Path to Digitization

We will explore how advancements in digital technologies are revolutionizing how organizations operate, necessitating the development of a strategy that outlines how these advancements will be leveraged to support the organizations' business strategy. Several trends will be examined, including intelligent process automation, data-centric AI, and business modular architectures.

We will also delve into the crucial role of Enterprise Architecture in defining and governing the IT strategy and roadmap, as well as their role in providing comprehensive system documentation and managing the portfolio of applications. Throughout, you'll gain an appreciation for how Process Inventory advances these trends by offering granular process information, enhanced data labeling, and alignment with business goals, enabling success in the digital age.

Netflix, Digital Leader

Netflix has emerged as a leader in embracing digital technologies to advance its mission, transform its business, and revolutionize the entertainment industry.

One of Netflix's most notable achievements is its successful transition from a DVD-by-mail sales and rental service to a streaming platform. Recognizing the increases in data speeds and the decreasing bandwidth costs, Netflix boldly disrupted its own business model by introducing a streaming service in 2007. This move allowed subscribers to access a vast catalog of movies and TV shows instantly over the Internet. By leveraging digital technologies, they eliminated the limitations and inconveniences associated with physical DVD rentals, paving the way for the streaming revolution [1].

A cornerstone of Netflix's success lies in its sophisticated use of machine learning algorithms to drive customer engagement. More than 80% of the shows people watch on Netflix are discovered through the platform's recommendation engine. Netflix has gathered hundreds of different attributes on each user based on their viewing behaviors [2] and categorizes them into different taste groups within their customer base. The recommendation engine not only personalizes title suggestions but also dynamically customizes the thumbnail of each show for each user to generate higher click-through rates. Additionally, Netflix extensively uses machine learning for other important use cases such as improving streaming quality, optimizing the show production process, and monitoring the quality of content.

Netflix has been a pioneer in transitioning to a cloud-based, loosely coupled, agile architecture. The company faced significant challenges due to the explosive growth of data and user information in its existing data centers. In 2009, they embarked on a transformation to break up their monolithic architecture into thousands of loosely coupled services, now known as microservices, deployed within a cloud infrastructure. The demand on their infrastructure is enormous, with Netflix apps consuming 15% of the Internet bandwidth worldwide and over six billion hours of collective viewership per day [3]. This microservices architecture has provided them with the ability to scale specific services on demand, accelerate software changes through an agile and continuous deployment model, and reinforce their environment with greater resiliency and fault tolerance. Since their success with this model, other prominent organizations have adopted this microservice architecture to address their own scaling challenges.

Through its strategic adoption of digital technologies, Netflix has achieved remarkable business success, growing from 857K subscribers and \$152M in revenue in 2002 to 209M subscribers and \$31.6B in revenue by 2022 [4]. As a digital pioneer, Netflix continues to push boundaries, redefine user expectations, and inspire organizations across industries to embrace the

power of digital technologies to transform their business models and deliver exceptional customer experiences.

Embracing Digital Technologies

The pace of technological progress is nothing short of breathtaking. Every day, new advancements emerge that promise to revolutionize the way we work and live. From process automation to AI technologies and cloud-native design, the potential for operational excellence is limitless. This is not a fad or passing trend, these digital capabilities will continue to progress exponentially. Those who fail to adapt will be left behind, while those who embrace these changes will thrive in today's rapidly evolving business landscape.

These trends will also have significant implications on the future of the workforce. As automation and AI become more prevalent, organizations will need to reskill their employees to work alongside these technologies. These should be viewed as productivity multipliers that enable employees to focus less on mundane and repetitive tasks and focus more on strategic and customer-facing value, which if done correctly will increase employee engagement and satisfaction.

To prepare for this, organizations will need to prepare, which means defining a digital strategy that aligns to their mission, detailing the use cases that will drive business value, enhancing their enterprise IT and data architecture design, and investing in workforce change strategies.

An imperative to success is to focus on excellence in the business process environment as this represents the intersection between the workforce and technology. The Process Inventory framework by being a comprehensive and accurate inventory of all the processes that an organization performs is a perfect enabler to ensure that all impacts are considered and addressed while maintaining the lens of business value.

In this chapter, I will cover the state of each of these topics and discuss how the Process Inventory framework solves the challenges faced today.

Intelligent Process Automation (IPA)

Process automation is a universally acknowledged imperative in digital transformation programs. According to a survey of IT decision-makers, 97% agreed that process automation is vital to digital transformation [5]. Process automation involves using technology to automate complex business processes, eliminating or reducing the need for human interaction. The technologies involved in process automation include robotic process automation (RPA), workflow, and intelligent automation, which leverages AI

technologies such as machine learning (ML), natural language processing (NLP), and computer vision.

When implemented correctly, process automation can be a game-changer that revolutionizes the way businesses operate with increased efficiency, reduced risk, and even revenue growth. According to a study conducted by McKinsey, two-thirds of the companies surveyed are actively pursuing process automation [6]. The market for intelligent process automation is forecasted to reach 30 billion US dollars by the end of 2024, with a compound annual growth rate (CAGR) of 16% [7].

However, it is estimated that 30%–50% [8] of automation projects fail. Here are the top reasons for failure:

- **Lack of Clear Goals and Integrated Governance:** Without clear alignment toward organizational goals and commitment from senior-level executives to sponsor the initiative, there can be challenges in securing sufficient funding and acquiring key subject matter resources. As a result, the program may struggle to gain traction, leading to subpar designs and incorrect results when implementing automation.
- **Incomplete Understanding of the Process Holistically:** Many efforts are focused on a task or part of a process within an organizational silos. The lack of understanding of the end-to-end process may lead to suboptimal design or challenges in handoffs from upstream or downstream processes.
- **Choosing the Wrong Process to Automate:** Processes that are too simple, too infrequent, or too changeable should not be automated as they may fail to have the desired impact needed to keep stakeholders engaged and may cause maintenance issues.
- **Choosing the Wrong Solution:** Automation teams can get locked into a single tool for all process automation needs. Choosing the wrong automation tool can lead to poor employee experiences or even codify a poor underlying system design.
- **Insufficient Training or Change Management:** Implementing automation can be disruptive to operational staff or create more issues if staff isn't properly trained and operating procedures aren't updated accordingly.

Defining the Intelligent Process Automation Program Charter

When embarking on an IPA program, it's important to have a clear strategy that outlines the goals and objectives of the program. A few key aspects of this strategy include the following:

A **business case** should be created to define how this effort will support the organization's strategy. It must identify the investment needed and the benefits to be gained. Since IPA is focused on improving human productivity, the benefits are usually stated in metrics such as increased productivity, reduced cost, improved process accuracy, improved customer experience, improved employee experience, and improved risk management. Part of the business case should include the design of a reporting dashboard and a plan for quantitatively capturing the resulting metrics.

The strategy should have clear details on the **scope of the effort and the selection criteria used to prioritize** automation opportunities. Factors such as complexity, repetitiveness, volume, error rate, cost savings, and risk ratings should be considered in a systematic manner. It's important to think through these details so that leaders have clarity when weighing different opportunities.

An **IT architecture strategy** is needed to provide clarity on which IPA tools will be available and for which use cases each should be leveraged (see Table 6-1). This should include a data management plan since intelligent automation requires quality data to be effective.

Table 6-1. Intelligent Process Automation Tools and Their Use Cases

Tool	Appropriate Use Cases
Robotic Process Automation (RPA)	RPA “bot” mimics human interactions with user interfaces to automate repetitive, rule-based tasks.
Workflow Automation	Workflow automation focuses on streamlining and automating the flow of tasks and activities within a business process. It ensures that tasks are assigned, executed, and monitored efficiently, often using a workflow management tool.
Artificial Intelligence	AI technologies enabling automation systems to perform cognitive tasks that traditionally require human intelligence. AI technologies used in IPA include machine learning, natural language processing, and computer vision.

These tools can be implemented stand-alone or combined depending on the complexity and the goals of each use case being addressed.

Several **operational model** considerations need to be addressed. First and foremost, senior-level sponsorship is critical. This leader or leaders need to understand the benefits and be passionate about the potential to be a champion. This sponsorship is important in securing necessary resources, gaining organizational buy-in, and clearing any hurdles.

The operating model for the program will have to be defined. There are options such as a centralized Automation Center of Excellence (COE) that performs all design and implementation work, or there are federated models where methods and standards are published, but each business unit or functional area performs their own automation. A methodology should be built that details the series of activities and deliverables, as well as business sign-off, to show how an automation opportunity goes from concept to implementation. Since many of these technologies are new, a plan will have to be identified on how to upskill existing staff or acquire skilled resources.

Overall, these steps and considerations can pave the way for a successful intelligent process automation program, driving efficiency and transformation within the organization.

Discovery and Assessment

Many Automation Centers of Excellence (COEs) typically reside in the IT organization, presenting a challenge as they may have expertise in automation tools but struggle to understand the process landscape.

Process documentation, as exemplified in Figure 6-1, plays a crucial role in supporting intelligent process automation efforts by providing a clear summary of existing business processes and serving as a foundation for automation scoping, analysis, and design.

It ensures that automation initiatives are built upon accurate and comprehensive knowledge not only of individual processes but also the context of those processes within the business, including upstream or downstream process dependencies. Process metadata offers rich information, such as the resources leveraged in process execution.

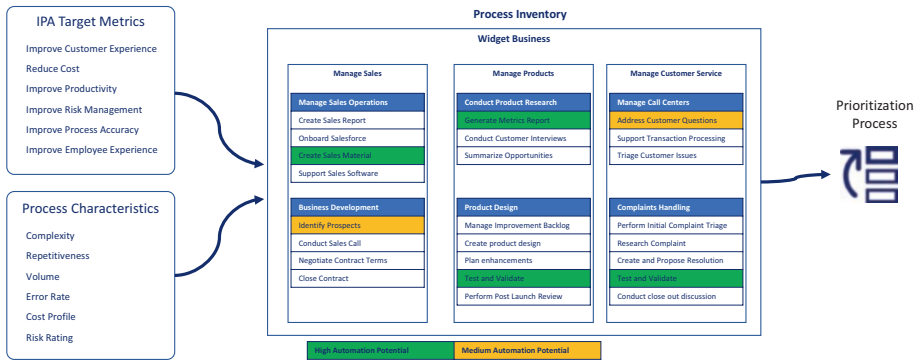


Figure 6-1. IPA Discovery and Assessment Process

The Process Inventory framework enables a **top-down identification of candidates**, which can start with interviewing the business unit or functional area leaders. They possess insights into their processes, particularly those causing the most pain, and can help align potential scope to their executive-level performance goals and metrics.

The assessment team can then collaborate with process owners and other key stakeholders identified in the Process Inventory framework to assess each opportunity. This involves conducting a deep dive into current performance metrics, process characteristics, and the details of process execution as captured in the Integrated Operational Repository. Based on this analysis, the teams can form hypotheses regarding the root causes of challenges, benefits, potential IPA solutions, and the required effort to implement an IPA solution.

The outcome of this assessment is a score reflecting the cost-benefits of each opportunity. A common summary, depicted in Figure 6-1 as a heatmap of opportunities, serves as a visual tool for engaging senior business leaders and stakeholders to gain consensus on which processes should be prioritized for automation.

Process Design and Optimization

Once a process is identified for automation, it is essential to redesign the process to fully leverage the capabilities of the automation technology.

During this phase, the design team builds upon the hypothesis developed during the assessment phase to determine the future state design. If a current state process model does not exist, one should be created, outlining the people, process steps, technologies, and controls that may be impacted by the design changes.

Using the principles of design thinking, the design team collaborates with process owners and other users to gain a detailed understanding of their pain points and challenges. By empathizing with the users, the team can select the most suitable IPA tool and create a future state process model, along with the necessary technical design documents.

At this stage, there is a comprehensive understanding of how the process redesign will impact the users. It is crucial to develop a change management plan that addresses potential disruptions. This plan may include training programs, updates to procedure documents, and communication strategies to ensure a smooth transition.

Furthermore, the business case should be refined to incorporate updated cost and benefit projections, as well as an updated implementation timeline. This ensures that the business case remains aligned with the evolving project requirements and goals.

Development and Testing

In this phase, developers take the future state designs to create the solution. Following an iterative development approach, the solution is developed in incremental stages, allowing for early feedback and adjustments based on user input. This iterative approach ensures a more user-centric and effective solution by actively involving end users in the development process.

Quality assurance and testing play a critical role in ensuring the reliability and functionality of the developed solution. Rigorous testing methodologies are employed to identify and resolve any issues or bugs. This includes various testing levels (which are covered more extensively in Chapter 5), such as unit testing, integration testing, and user acceptance testing. Unit testing focuses on testing individual components or modules of the solution, while integration testing verifies the interaction between different components. User acceptance testing involves end users validating the solution against predefined criteria to ensure it meets their requirements.

Additionally, a pilot implementation is conducted with a subset of users to validate the expected benefits and gather further feedback. This real-world testing provides an opportunity to fine-tune the solution, validate its performance, and make any necessary refinements before rolling it out to the entire user base.

As part of the development and testing phase, it is essential to update documentation to reflect the changes introduced by the automation solution. This includes updating the Process Inventory, creating a new process model, revising control documentation, and updating procedure documentation to reflect any changes in tasks or responsibilities resulting from the automation implementation.

Data-Centric AI

The advancement that holds the greatest potential for disruption is machine learning models, a branch of artificial intelligence that utilizes data to enhance computer performance by enabling machines to “learn.” In modern-day machine learning, there are two primary objectives. The first is to classify data based on developed models, while the second is to make predictions for future outcomes using these models [9]. Machine learning has been a concept since the 1950s but has recently gained significant attention with the release of ChatGPT in November 2022 and its impressive capabilities in natural language processing and conversation generation. ChatGPT, which is a generative AI technology that leverages machine learning models, is generating excitement for what it can mean for productivity.

The potential for this technology to drive business performance is enormous. According to a McKinsey report [10], generative AI has the potential to contribute an estimated \$2.6 trillion to \$4.4 trillion annually across the 63 use cases they analyzed. Machine learning models have been successfully deployed in various use cases, including customer service chatbots, personalized marketing, and more. However, the level of success with this technology varies, with digitally native technology companies leading the pack, while traditional companies are investing to catch up.

Even for leaders in this field, the full potential of AI technology has not yet been fully realized, indicating that there is still a long way to go. The true value of AI emerges when it is operationalized at scale. Business leaders seeking to maximize the benefits of AI understand that scale refers to the deep and wide integration of AI into an organization’s core products, services, and business processes [11].

Renowned computer scientist and entrepreneur Andrew Ng has championed a movement called data-centric AI, which he believes is the key to unlocking additional value [12]. According to Mr. Ng, the recent focus in AI has primarily been on advancing machine learning models and software code, which have now reached a mature stage. However, the limiting factor in realizing added value lies in the quality of data.

Data is the fuel for a machine learning system as it relies on data for analysis and pattern recognition. To illustrate its importance, ChatGPT 4, for instance, boasts an estimated 100 trillion parameters [13], with parameters being mathematical relationships that connect words through numbers and algorithms. In the emerging field of MLOps (machine learning operations), which involves operationalizing machine learning models, up to 80% of the effort is dedicated to ensuring data quality [14].

According to Ng, a data-centric approach involves systematically engineering high-quality data for these models to consume. High-quality data is characterized by consistent labeling. Data labeling involves adding one or

more informative labels to provide context for machine learning models to learn from. For example, if different groups within an organization use diverse terms like “Payment,” “Wire Payment,” and “Move Money” to describe the same process, although none of these terms are inherently incorrect, the lack of consistency poses a challenge for the models to establish connections between events, ultimately leading to limited insights.

State of Data Quality

According to a study conducted by KPMG, 84% of CEOs are concerned about the quality of the data they’re basing decisions on. Gartner measures that the average financial impact of poor data on businesses at \$9.7 million per year [15]. Trust is key; if business leaders can’t trust the insights from these models because the underlying data is flawed, then efforts to more broadly adopt machine learning may struggle.

Data governance is the function that is responsible for ensuring data quality through all its life cycle stages. The problem is that large-scale data governance programs that live up to this mandate are rare. According to a study conducted by Gartner, only 42% of organizations are succeeding in optimizing data for business processes and productivity [16]. This statistic just underscores that leaders can’t wait for data governance programs alone to address this situation.

Even if data quality could be addressed through effective data governance, there are more challenges. A typical data environment consists of heterogeneous and unconnected data sources and unstructured data which are managed in organizational silos. When data is siloed within individual departments—an all-too-common occurrence—companies will struggle to realize generative AI’s true potential [17].

Process Inventory Provides an Enterprise Labeling Schema

In order to fully leverage the potential of machine learning, which involves integrating it deeply and extensively into an organization’s core products, services, and business processes, resulting in operational excellence characterized by optimized efficiency and maximized resource utilization, it is essential to have a consistent labeling schema across the entire enterprise and all AI use cases.

Process Inventory serves as the foundation for this consistent labeling schema. As explained in Chapter 2, an organization’s comprehensive knowledge is captured through an ontology, which in this case is a polycategorical ontology due to the integration of multiple concepts with their respective taxonomies.

The fundamental taxonomy in this context is the Process Inventory as all concepts, except for strategy, rely on processes for their existence. Therefore, the Process Inventory acts as the unified language that connects all concepts together coherently.

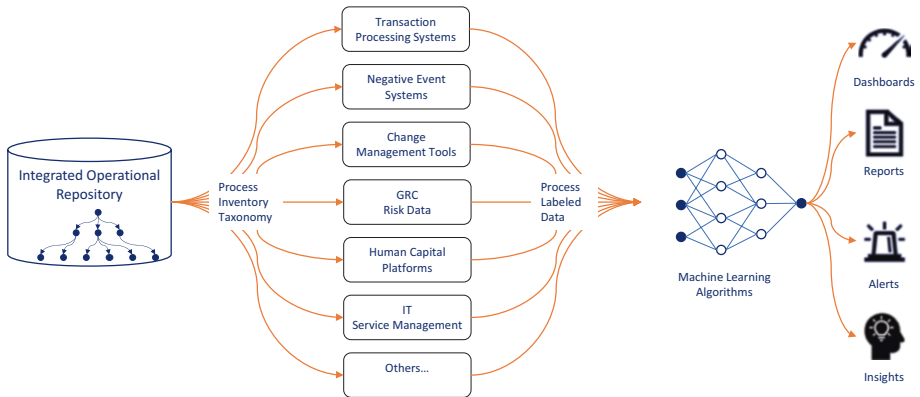


Figure 6-2. Process Inventory Enterprise Labeling Schema

Figure 6-2 illustrates a solution to this problem. The Process Inventory taxonomy is published to multiple systems responsible for generating and managing data related to the organization's core products, services, and business processes. These systems align their data with the corresponding processes in the Process Inventory taxonomy. For instance, GRC systems would label risks based on the associated processes (as described in Chapter 7), while transaction processing systems would label customer behavior data according to the process that drove the interaction.

Using a consistent labeling schema promotes transfer learning, which is where knowledge learned from one use case can be applied to other use cases. This enables improved performance of models by reusing labeled data. By applying knowledge from one use case to the next can uncover hidden patterns that might have been overlooked otherwise. For instance, data collected to identify emerging risks in order to identify and strengthen controls can detect patterns of resource usage which can lead to optimization of costs for certain processes.

This will fundamentally change how businesses are run and the relationship between humans and machines within organizations. Competitive advantages will go to the early movers who define their strategy and move toward creating the data environment required for success.

Use Cases for Operational Excellence

The potential impact of AI on transforming business operations cannot be overstated. Through the implementation of an enterprise labeling schema, machine learning models can gain access to all aspects of a business within a single, comprehensive indexed framework. This framework enables a holistic understanding of the organization, empowering the models to make informed decisions and provide valuable insights across various business functions.

The **strategic decisions** leaders make can be transformed as AI can process vast amounts of internal and external intelligence to provide a thorough assessment of threats and opportunities. It can simulate multiple scenarios to assess potential outcomes of strategic decisions. This capability gives organizations a competitive edge in strategy formulation and decision-making.

Customer experience can undergo a transformation, not only through customized and personalized content that leverages a 360-degree view of the customer but also with predictive customer service. This involves anticipating potential issues or needs and proactively addressing them. Continual monitoring of sentiment analysis and customer feedback allows for the identification of opportunities to enhance the customer experience and pinpoint the root causes, enabling teams to address them promptly.

A new level of **operational efficiency** can be achieved by monitoring near-real-time performance data to identify where business processes and the use of resources can be optimized. By including information on upstream and downstream partners, the supply chain can be optimized to streamline logistics, refine inventory levels, and reduce lead times.

Change management investments become more effective as discretionary change dollars are allocated to the highest level of strategic need. Impact analysis for proposed changes becomes precise in terms of identifying what needs to change and how. This improvement leads to more accurate budget and timeline estimates, as well as driving new levels of accountability for implementation. Additionally, personalized communication and training can be delivered to employees who are impacted by the change, facilitating a smooth change deployment. This can include targeted communication campaigns, training modules tailored to specific roles, and change management support to address employee concerns.

Risk management efforts create a more comprehensive and quantifiable picture of the risk landscape, providing senior leaders and the board with increased confidence in making strategic decisions. AI can analyze vast amounts of data related to each process, enabling the identification of emerging risks and the evaluation of the effectiveness of existing controls. Compliance becomes more accurate as changes to laws and regulations are monitored, allowing for the identification of necessary adjustments to

compliance processes and controls. AI-powered calculations of financial risks are enhanced, potentially eliminating or significantly mitigating substantial losses. For example, AI algorithms can analyze financial market data, identify patterns, and assess the likelihood of potential losses, enabling proactive risk management measures.

Engagement in the workforce rises significantly as repetitive and mundane work gets replaced by more strategic and value-added activities. Knowledge and training become more readily accessible and personalized to the role and needs of each employee.

It's imperative that organizations formally create strategies for adopting these new technologies. This includes identifying a prioritized set of use cases, defining a new operating model, creating policies to address the risks these platforms pose, establishing quality guidelines, and designing an architecture and vision for safely integrating this technology into their environment.

AI technologies are here to stay! The urgency lies in establishing a strategy and tactics to adopt this technology to stay competitive or even thrive in this new world.

Business-Led Modular Architectures

Traditionally, IT has often relied on a monolithic architecture paradigm in which all the logic for a broad set of functionalities is contained within a single, often large code base. In monolithic architectures, the application's components, including the user interface, business logic, and data storage, are tightly interconnected and interdependent. Consequently, making changes to functionality requires extensive coordination among various stakeholders and rigorous regression testing to prevent unintended impacts on other processes. Even minor changes could take months to deploy, while major changes often require years of effort and significant costs before reaching production.

In today's fast-paced digital world, the ability to respond quickly to customer expectations and competitive pressures is paramount. This necessitates an agile IT architecture that allows businesses to adapt rapidly and deploy changes within much shorter timelines. Given the significant dependence on IT systems for business functionality, a new paradigm has emerged: a business-led, modular, and loosely coupled IT architecture. "Business-led" means that IT services directly align with the goals and processes of the business, enabling greater involvement of business stakeholders in shaping system functionality to meet their needs. "Modular and loosely coupled" means that software is deployed in independent and decoupled components or modules, allowing for flexibility, scalability, and easy maintenance. Additionally, organizations are increasingly migrating their systems to the cloud and adopting cloud-native architectures, which enhance scalability, fault tolerance, flexibility, security, and cost savings.

The trend in IT architectures toward modular architectures dates to the advent of service-oriented architectures (SOA) which gained popularity in the 1990s. The concept was to break apart large monolithic systems into smaller components, which would be accessed through API-based interfaces that represented the business function being requested. SOA was a huge success if you measure it by adoption, which was widespread. It certainly was a step in the right direction as SOA services would be aligned to the business functionality that was accessed, which increased reuse, interoperability, and in some cases, agility. But this had limitations as SOA services, in many cases, were facilitated by middleware applications, but the underlying architecture remained monolithic, so some of the underlying challenges remain.

The next evolution of this paradigm, increasingly popular in many scenarios, is the microservices architecture. This was popularized when Netflix, driven by the increasing data and user information which was difficult to manage in their current data center infrastructure, was driven to this new architecture. Since then, Google, Amazon, Twitter, and others just to name a few have embraced this concept.

Microservices [18] are an architectural and organizational approach to software development where software is composed of small independent services that communicate over well-defined APIs. This means that large monolithic systems, in this new style, would be replaced by many small-grained microservice applications which are independent. At the current estimate, Netflix has over 1,000 microservices [19] each managing a different part of their site.

Here are some characteristics of the microservices architecture:

- **Loosely Based Autonomous Services:** Each service is stand-alone, meaning that each has its own business and data access logic and only communicates with other microservices via well-defined interfaces.
- **Independently Deployable:** Each microservice can be built, tested, and deployed independent of other microservices.
- **Aligned to Business Capabilities:** Microservices are designed with the single-responsibility principle, meaning they are to have one responsibility and do that responsibility well. A business capability model is employed to define and rationalize these responsibilities across the business and to establish the bounded context for each microservice.

This has revolutionized how software is developed. Since services are independently deployable and cover a small-bounded context of business functionality, that means that they can be **owned by small cross-functional feature teams**. Ownership means everything from development, deployment, and maintenance for that specific business functionality. Cross-functional teams include developers, testers, process, and product owners.

This removes the significant amount of coordination across a large set of stakeholders that comes through monolithic architectures. This has led to the integration of the development teams with the IT operations teams through DevOps which has enabled software development to be truly Agile. This is accomplished through continuous integration and continuous delivery (CI/CD) which is a set of practices, tools, and methodologies to automate the building, testing, and deployment of software. This has enabled Amazon, one of the most technically complex companies around, to average a deploy of software every second to the development, testing, and production environments [20].

Microservices are designed to be cloud native which gives them all the benefits of the cloud such as scalability on demand, resiliency, flexibility, and security.

Anchoring Microservice Design to Process Inventory

According to a survey conducted by Camuda [21], the top challenges in implementing microservices include the lack of visibility into end-to-end business processes spanning multiple microservices (59%), error handling issues at the boundary of two or more microservices (50%), and communication between teams (46%).

While aligning designs to a business capability model has been beneficial for the microservices community, the characteristics of a business capability model being organizationally agnostic and not linking to business processes have proven limiting. This presents challenges in demonstrating the processing idiosyncrasies across business units and capturing the necessary level of detail for showcasing end-to-end processes.

Process Inventory equips design teams with comprehensive details to enhance their designs. By combining the Process Inventory with customer journeys or value stream maps, teams gain an **end-to-end view** of how processes support customers or internal stakeholders, fostering seamless integration and interoperability between microservices.

As a top-down derived taxonomy, the Process Inventory establishes clear boundaries for **defining the bounded context** of microservices, ensuring a focused and well-defined scope, leading to a rationalized and optimal set of services.

The **metadata associated with processes**, including intricate data details, aids in designing precise API interfaces for microservices. Additionally, events identified in process models play a crucial role in microservices design. Events serve as triggers that call a microservice interface, facilitating communication between microservices. They also signify state changes for data entities. For example, when the event “LoanApproved” occurs, the data table storing the loan data will transition from “In Process” to “Approved.” Moreover, microservices are complementary to event-driven architectures as a process-aligned service can publish events upon completing its work, thereby triggering other microservices.

Process ownership, as specified in the Process Inventory, establishes accountability, incorporating valuable business stakeholder input and approval. This also helps in defining feature teams to ensure they have the appropriate ownership composition.

Furthermore, this structured approach facilitates microservice reuse as the Process Inventory and business capability models enable teams to identify and integrate requirements from similar processes across diverse business units.

Figure 6-3 illustrates this approach with a Process Inventory for a retail bank. The processes detailed in the diagram form the basis for microservice design as each process will have its own activities and processing requirements from the underlying IT systems.

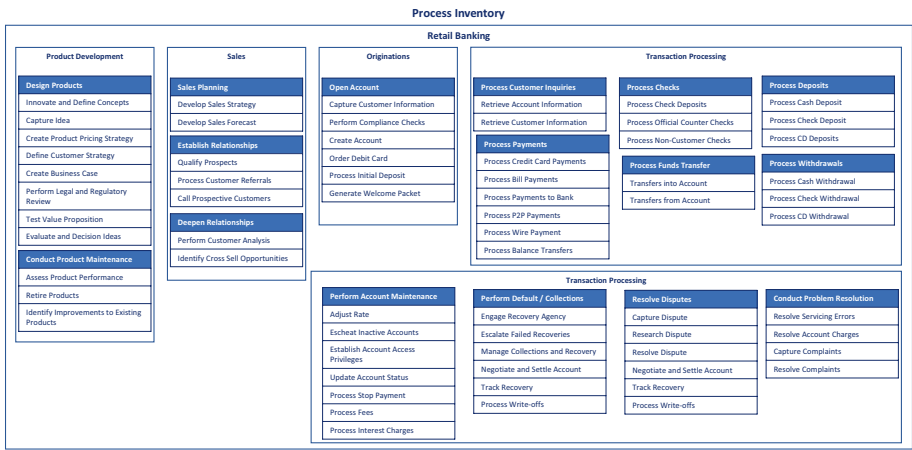


Figure 6-3. Banking Process Inventory

Enabling Enterprise Architecture Design

Prior to investing in any of the aforementioned technologies, it's imperative for an organization to create an IT strategy, target state vision, and roadmap that supports the business's ambitions. This responsibility typically falls under the mandate of an Enterprise Architecture organization, which reports to the Chief Information Officer (CIO). Enterprise Architecture is the process by which organizations standardize and organize their IT environment to align with business goals. These strategies support digital transformation, IT growth, and the modernization of IT [22]. Among their responsibilities are building and governing the journey to the target state vision.

This task is not an easy one. Enterprise Architecture often faces a natural tension with business leaders who are confronted with immediate business needs and cannot afford the cost or timelines associated with implementing technology solutions that align with the long-term IT vision. I have witnessed this scenario play out multiple times during mergers, where the communicated timeline to investors did not allow for the rationalization of redundant applications. As a result, end users had to work with different systems depending on the customer base they were servicing.

Other pressures, such as a poorly conceived IT vision, inadequate business requirements, insufficient technical documentation, and a tightly coupled IT architecture, contribute to the challenges. These factors lead to the accumulation of technical debt, which refers to the implied cost of reworking suboptimal solutions at a later stage. According to a recent McKinsey survey, CIOs reported that 10%–20% of the technology budget allocated to new products is diverted to resolving issues related to tech debt.

Consequently, many organizations end up with a complex and fragmented IT landscape characterized by redundant systems, poor interoperability, legacy technology platforms, and variations in architectures. The proliferation of end-user computing, which involves the development and management of applications outside the central IT organization, further exacerbates this complexity and fragmentation.

These environments become a web of dependencies that stifle innovation, reduce agility, increase security vulnerabilities, undermine resiliency, and hinder the organization's ability to meet customer and market expectations. In severe cases, complexity can degrade customer experience and pose significant risks to the organization, which are contrary to the requirements for successful digital transformation initiatives.

Enterprise Architecture Frameworks

There are many different Enterprise Architecture frameworks such as The Open Group Architecture Framework (TOGAF), Zachman Framework, Federal Enterprise Architecture Framework, and more. These frameworks help structure architects' thinking by identifying subdomains from which they can align organizational objectives. These domains are consistent across these frameworks, which include [24]

- **Business architecture** defines the business strategy, governance, organization, and key business processes of the organization.
- **Information/data architecture** describes the structure of an organization's logical and physical data assets and the associated data management resources.
- **Application architecture** provides a blueprint for the individual systems to be deployed, the interactions between the application systems, and their relationships to the core business processes of the organization with the frameworks for services to be exposed as business functions for integration.
- **Technical architecture** describes the hardware, software, and network infrastructure needed to support the deployment of core, mission-critical applications.

The value and understanding of the business architecture domain will be enhanced through the incorporation of the Process Inventory, which complements existing artifacts like the business capability model by providing additional details on business processes.

In the following sections, I'll cover key Enterprise Architecture responsibilities and how Process Inventory supports the effectiveness of their efforts.

Translate Business Strategy into IT Strategy

A business strategy defines the initiatives an organization will undertake to fulfill its purpose and create value for the organization and its stakeholders. The business strategy incorporates components that an IT strategy should anchor to, such as product differentiation, customer strategy, critical business capabilities, metrics goals, and in a digital context, how the organization plans to maximize the capabilities of digital technology.

The IT strategy, on the other hand, should focus on how IT will enable business success. During the development of the strategy, the Enterprise Architecture team will need to make a series of trade-off decisions, such as buy vs. build,

insource vs. outsource, tactics for agility, data integration style, and more. These decisions are typically formalized through IT principles, which are widely distributed to guide teams in making design decisions. The IT strategy should also outline operational considerations, including governance and financial management structures required to achieve the strategic vision. Additionally, the strategy should identify success metrics that align with the business strategy.

Process Inventory plays a crucial role in bridging these two strategies by providing a comprehensive view of an organization's business processes. It enables a deeper understanding of how technology can align with and support those processes. This understanding is essential for aligning success metrics, ensuring that the IT organization aligns its goals and performance measures with the strategic objectives of the business.

Create Transparency

In my experience, many IT organizations have inadequate documentation on their systems and the interactions between systems that support business process activities or serve other purposes. This lack of transparency results in numerous inefficiencies as the reliance on a few key subject matter experts becomes necessary for troubleshooting issues, identifying risks, analyzing the impact of changes, and performing other tasks. Additionally, this situation creates a key person risk as the departure of these experts could lead to a loss of critical knowledge.

The challenge has been the absence of consistent standards for modeling and governing these interaction models and mapping them to the business context. Solution architects often create diagrams during the design phase of a project, but once the project is deployed, these diagrams are rarely updated.

Since Process Inventory provides a complete and accurate inventory of all business processes in the organization, it delivers a comprehensive organizing framework for aligning IT assets to the business processes they support. It goes further: process models are created to illustrate how business processes are executed, providing a platform to demonstrate the flow of data through underlying systems.

CAPITAL MARKETS ORGANIZATION

This Capital Markets Organization had become concerned about the extensive use of legacy mainframe COBOL for executing high-volume critical transactions. The potential consequences of system failure were deemed catastrophic, highlighting the resiliency risk associated with relying on a limited number of subject matter experts. Their objective was to map these transactions with the goal of migrating the functionality to a modern cloud infrastructure.

To address this challenge from a business perspective, the organization established a business architecture team. They sought a partner who could support their new team in adopting an appropriate approach.

The project spanned 16 weeks and focused on the following areas: creating a Playbook with detailed standards and methods, documenting the Process Inventory across the enterprise, identifying requirements for a modeling tool, and executing proof of concepts to map the rules within the COBOL transactions.

Within this timeframe, we successfully mapped the Process Inventory and proceeded to decompose one set of processes to illustrate how human activities in a business process triggered interactions and business rules within the COBOL platform. This accomplishment provided the necessary foundation for the team to continue with the remaining key processes and define requirements for their cloud solution.

A component of the Unified Modeling Language, **system sequence diagrams (SSD)** are not a new concept. What is new is linking them to the inventory of processes in the organization. Linking process models to SSDs provides traceability between activities of the business and actions of the underlying systems, which is critical for business alignment. It provides a structure for maintaining these diagrams over time which will drive efficiency in all processes related to the IT ecosystem.

Figure 6-4 shows an example of how an activity in a business process can trigger a system event to be captured in a system sequence diagram.

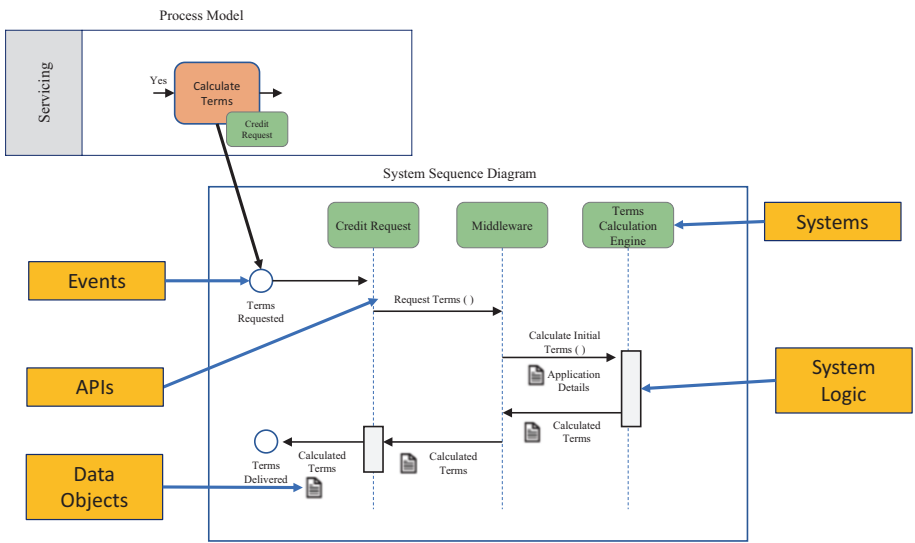


Figure 6-4. System Sequence Diagram Example

All actions within an organization, including those in the IT environment, start with a triggering event. For the purpose of categorization, events can be broadly classified as business events, which are triggered during the fulfillment of a business process, or system events, which are triggered by system actions such as scheduled batch jobs. Business events are represented in process models, and since these processes are documented in the Process Inventory, it provides an ideal mechanism for alignment with the business context.

Each activity performed on a system within a business process represents a system event that initiates the system sequence diagram (SSD). In the given example, an activity called “Calculate Terms” triggers an event called “Terms Requested,” which in turn kicks off the SSD. Swimlanes in SSDs represent systems, and each line entering a system represents an API for that action. The blocks within each lane depict the logic performed by each system. In cases where the logic is complex or supports a critical process, details on business rules or logic can be detailed. Data objects are also represented to demonstrate the flow of data and data lineage.

The creation and maintenance of these diagrams can be facilitated using an Enterprise Architecture modeling tool. Several vendor tools provide business process and Enterprise Architecture modeling capabilities within the same tool. These tools treat everything as objects, which is important for reporting and impact analysis purposes. Establishing a new team within Enterprise Architecture to build and maintain these models, mirroring the roles and responsibilities of the Process COE, will be required.

The power of this structure cannot be emphasized enough. Integrating business and technology models allows the organization to be viewed from a top-level C-suite perspective and drilled down to the granular level of logic within a system. This alignment ensures that everything within the IT ecosystem is directly tied to the business activities they support.

Building and maintaining these models will require investment and commitment, but the payoff in terms of increased efficiency, reduced risk, decreased security vulnerabilities, and enhanced innovation will be substantial.

Application Portfolio Management

One of the important roles of an Enterprise Architecture team is managing the portfolio of applications or systems within an organization. As complexity grows, organizations may end up with multiple systems that perform similar or redundant functions. This complexity increases the costs and risks associated with IT.

APM is a governance activity that involves building and maintaining an accurate inventory of systems. Each system is assessed based on its functional fit, which determines how well it serves the business, and its technical fit, which assesses

how the system aligns with the IT strategy. The outcome of this assessment is typically a determination of the organization's position on the system, which could be categorized as buy (to expand the use of existing systems), sell (to explore opportunities for retiring or divesting systems), or hold (to maintain the system without significant changes).

By conducting application rationalization projects through application portfolio management (APM), organizations can effectively reduce redundancy, streamline their application portfolio, and align their IT investments with business objectives. This process not only optimizes costs but also mitigates risks and enhances the overall efficiency and effectiveness of the IT landscape.

During the functional fit assessment, applications are labeled according to the business capabilities they serve. This labeling helps identify overlaps and redundancies within the application portfolio. Additionally, aligning systems based on business processes, as documented in the Process Inventory, adds a level of granularity that aids the assessment team in governance and decision-making processes. The inclusion of business process alignment allows for a more detailed analysis of system dependencies and impacts, facilitating thorough impact analysis when retiring or decommissioning a platform. This level of granularity also assists in documenting detailed requirements for application rationalization projects.

By integrating the information from the Process Inventory and the functional fit assessment, organizations can make well-informed decisions about retiring, consolidating, or modernizing applications. This holistic approach to application rationalization helps ensure that the resulting portfolio of applications aligns with business processes and requirements, enabling organizations to achieve their strategic objectives more effectively.

Define Target State Vision and Roadmap

After the creation of the IT strategy, the Enterprise Architecture team will collaboratively develop a target state vision that aligns with the organization's business strategy and objectives. This vision provides a forward-looking perspective, typically spanning three to five years, and encompasses various aspects such as changes to the integration architecture, adoption of emerging technologies, new methods or frameworks, reengineering of key process areas, and modifications to the IT operating model.

To ensure the target state vision is well-aligned with the business, the Enterprise Architecture team actively engages key stakeholders, including business leaders, process owners, and IT teams. This collaborative approach facilitates a deeper understanding of business needs, challenges, and priorities. By involving stakeholders in the process, the team can incorporate their insights and ensure the target state vision reflects the organization's strategic direction.

As part of the target state vision development, the Enterprise Architecture team conducts a comprehensive gap analysis to identify the initiatives necessary to bridge the current state and the desired future state. While these initiatives are defined at a conceptual level rather than detailed requirements, the availability of Process Inventory enhances the precision in identifying impacts, costs, and benefits. The granular view provided by Process Inventory enables the team to assess the implications of proposed changes on specific processes, identify potential overlaps or gaps, and estimate the resources required.

Moreover, the Process Inventory remains foundational throughout the refinement of estimates, prioritization of initiatives, and the construction of business cases to secure funding. By continuously leveraging Process Inventory, the Enterprise Architecture team can refine and validate their understanding of the processes, identify further optimization opportunities, and ensure that the initiatives align with the overall business strategy.

Through a collaborative and iterative approach, the target state vision, supported by the insights gained from Process Inventory, enables organizations to align their IT initiatives with business objectives, gain stakeholder buy-in, and pave the way for successful execution.

Key Takeaways

- **Digital technologies** are advancing at a rapid pace and revolutionizing how organizations operate. It's imperative for organizations to define a strategy for leveraging these advancements to further their business goals.
- **Key digital technologies** that organizations are deriving business value from include intelligent process automation, data-centric AI, and business-led modular architectures.
- **Enterprise architecture** plays a pivotal role in defining and governing the IT strategy and roadmap. Additionally, it is responsible for overseeing system documentation and managing the application portfolio.
- **Process Inventory** is a critical enabler for all these efforts, providing alignment across business goals and IT designs at the granular process level.

In the next chapter, we will delve into how the Process Inventory framework can enable stronger risk management.

Strengthening Risk Management

In this chapter, we will explore the growing importance of risk management in today's complex business landscape. We'll discuss the factors driving this importance, including increasing complexity, regulatory requirements, and the evolving role of risk oversight in boards.

Throughout the chapter, you will gain insights into the components of an Enterprise Risk Management program. We'll delve into various opportunities to enhance risk management, such as the adoption of quantitative measurements, minimizing human bias in risk assessments, improving cross-organizational coordination through end-to-end process awareness, enhancing risk data quality, and clarifying the risk accountability model.

Furthermore, we will examine how Process Inventory can play a pivotal role in strengthening risk management by aligning all risk-related activities with processes. This approach has the potential to elevate the quality of risk data, strengthen the risk operating model, and leverage emerging technologies to

elevate risk intelligence. We will review examples in the areas of business continuity, compliance, and operational risk to illustrate how these concepts work in practice.

Deepwater Horizon Oil Spill

At 7:45 p.m. CDT on April 20, 2010 [1], high-pressure methane gas from a well that the Deepwater Horizon was drilling expanded into the marine riser and rose into the drilling rig, igniting and causing an explosion that engulfed the platform. Eleven workers went missing and were never found, while 94 other crew members were rescued by lifeboat or helicopter. The Deepwater Horizon drilling rig, located approximately 50 miles off the Louisiana coast, sank on the morning of April 22. The rig wouldn't be declared sealed until September 19 of that year.

Approximately 210 million gallons of oil were spilled into the Gulf of Mexico [2]. According to satellite images, the spill directly affected 70,000 square miles of ocean, which is comparable to the size of Oklahoma. The incident disrupted the coastal economy and devastated the ecosystem, resulting in the deaths of as many as 105,400 seabirds, 7,600 adult sea turtles, and 160,000 juvenile sea turtles, as well as up to a 51% decrease in dolphins in Louisiana's Barataria Bay [3].

The fallout for British Petroleum (BP) was significant. In November 2012, BP and the US Department of Justice settled criminal charges, with BP pleading guilty to 11 counts of manslaughter, two misdemeanors, and a felony count of lying to Congress. BP and the DOJ agreed to a record-setting \$4.5 billion in fines and other payments [4]. BP was also responsible for the cost of the cleanup, which was estimated to be more than \$65 billion [5]. In September 2014, a US district court judge primarily held BP responsible for the oil spill due to gross negligence and reckless conduct, leading BP to agree to pay \$20.8 billion in fines [6].

The Deepwater Horizon incident represents a failure of risk management within BP. A quote from BP's Accident Investigation Report stated the following: "Through a review of rig audit findings and maintenance records, the investigation team found indications of potential weaknesses in the testing regime and maintenance management system for the blowout preventer (BOP)" [7].

Importance of Risk Management

This is one of several recent high-profile failures in risk management, including the 2008 financial crash, the Boeing 737 crashes, the Fukushima nuclear disaster, and the Enron accounting scandal, to name a few prominent ones. It underscores the need for strong risk management, especially when customer, market, or environmental concerns are at stake.

In today's digital age, there are several factors driving organizations to implement a formal risk management program. The environments in which organizations operate are increasingly complex, with interconnected supply chains, globalization, and technological advancements. Emerging risks, such as cyber threats, ESG mandates, geopolitical instability, natural/weather events, and most recently, a global pandemic, are present. Regulatory requirements have been growing exponentially. According to the US Chamber of Commerce, "Over the last 60 years, the U.S population has increased by 98 percent, while the federal regulatory code has increased by 850 percent" [8].

Beyond surviving and managing the downside of risk, organizations increasingly view risk management as strategically important and critical in decision-making. According to a Deloitte survey [9], companies with a compound annual growth rate (CAGR) of 5% or more were twice as likely to consider risk management key to achieving strategic goals compared to those with a CAGR under 5% (40% vs. 20%).

Risk management is vital in corporate governance. The US Securities and Exchange Commission (SEC) requires boards of publicly traded companies to provide risk oversight. The SEC emphasizes the importance of effective risk management and expects boards to fulfill their responsibilities in overseeing the company's risk management processes.

Shareholders are increasingly holding board directors accountable for their role in risk oversight, as highlighted by examples from a speech by the former director of the SEC, Luis A. Aguilar [10]:

In early 2014, in the wake of a coal-ash spill, the California Public Employees' Retirement System ("CalPERS") and the New York City Comptroller urged shareholders to vote against four independent directors to Duke Energy Corp.'s board, accusing them of having "failed to fulfill their obligations of risk oversight as members of a committee overseeing health, safety and environmental compliance at the company." Separately, in 2013, a prominent shareholder advisory firm recommended that shareholders withhold support for three directors of the board of JPMorgan Chase and Co., accusing them of "material failures of stewardship and risk oversight" after the bank suffered a multibillion-dollar loss as a result of the "London Whale" trading scandal.

With the board, shareholders, and other stakeholders taking an increasing interest in how risk is managed, it is important to establish a formal risk program that provides a comprehensive view. This enables strategic and tactical decisions to be informed appropriately, prompting many organizations to develop a formal Enterprise Risk Management program.

Enterprise Risk Management

The Committee on Sponsoring Organizations of the Treadway Commission (COSO), a prominent risk management standards organization, defines Enterprise Risk Management (ERM) as a process, effected by an entity's board of directors, management, and other personnel, applied in strategy setting and across the enterprise, designed to identify potential events that may affect the entity, and manage risk to be within its risk appetite, to provide reasonable assurance regarding the achievement of entity objectives [11].

The primary goal of ERM is to enable organizations to effectively anticipate and respond to risks in a holistic manner, considering internal and external factors that may influence desired outcomes. Risks come in various types, which will be covered in the following section “Risk Taxonomy,” and may require specific approaches in the risk management process. Additionally, nearly every part of the organization can be involved in identifying and mitigating risk, making a holistic approach a significant undertaking.

COSO and the International Organization for Standardization (ISO) 31000 Risk Management Process are two leading sources of risk management guidance. According to a 2020 ERM Benchmarking Study conducted by the Risk & Insurance Management Society (RIMS) with 288 organizations, 37% of organizations have custom-built their own frameworks based on various sources [12], including these two organizations.

Figure 7-1 presents an ERM framework that incorporates standards from COSO, ISO 31000 Risk Management Process, and Management of Risk: Guidance for Practitioners (Orange Book). This framework serves as a reference point to discuss the structure of a robust ERM program, address current challenges, and illustrate how the Process Inventory framework can enhance the effectiveness of these efforts.

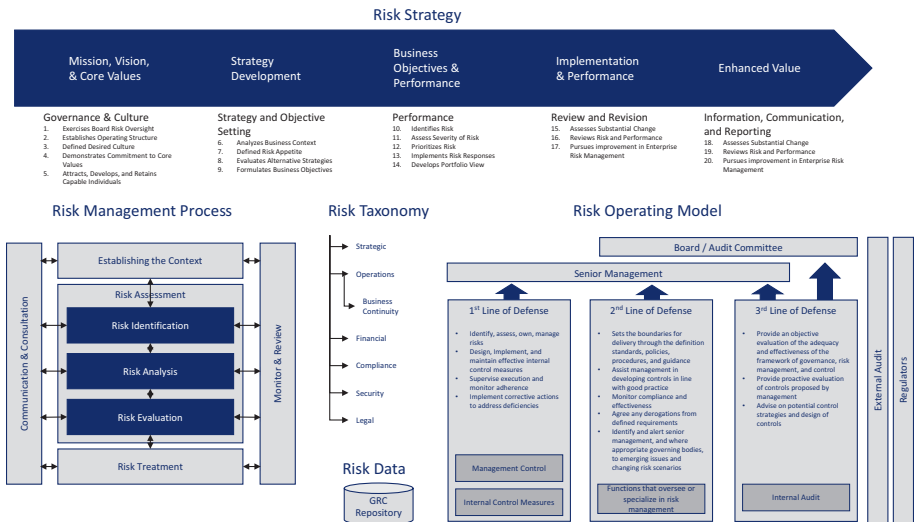


Figure 7-1. Enterprise Risk Management Framework

A strong ERM program typically exhibits the following characteristics:

- Integrated Approach:** Risk management should be seamlessly integrated into the organization's culture, operations, and governance structure. It should be an inherent part of strategic planning and decision-making processes.
- Governance:** Various stakeholders within the organization should have defined roles, responsibilities, separation of duties, and accountability for risk management. Senior management and the board must prioritize their oversight role and ensure clear documentation of their responsibilities.
- Risk Assessment:** Risks should be systematically identified, assessed, and prioritized across the organization. Since different types of risks exist, each may require a unique assessment approach. This typically involves rating the likelihood and impact of risks using a risk matrix.
- Risk Appetite:** Organizations need to define their tolerance for risk in alignment with their strategic objectives. This establishes a framework for prioritization and strategic decision-making processes.

- **Risk Treatment:** Effective risk response strategies should be developed to adequately control and manage identified risks.
- **Communications and Consultation:** Accurate and timely reporting on risks is crucial, especially for the board and senior leaders. Key stakeholders should receive appropriate risk information, and risk managers should be available to provide guidance and consultation in decision-making processes.
- **Monitor and Review:** Continuous monitoring and review of risks are essential to validate accurate risk identification and assess the effectiveness of implemented controls in mitigating risks.
- **Continuous Improvement:** The organization should foster a culture of innovation by regularly evaluating the performance of the risk management program and identifying opportunities for improvement.

Opportunities in Enterprise Risk Management

The history of risk management dates to ancient practices. Merchants in Babylon would distribute their goods across different ships to reduce the risk of losing everything in a single event [13]. Much of the history is tied to insurance, where in maritime cities such as Genoa and Venice, merchants and shipowners started pooling their resources to protect against the risks of shipwrecks, piracy, and other perils of the sea.

The emergence of life insurance and the need for assessing mortality risks in the eighteenth century led to the origins of actuarial science, applying mathematical principles to analyze life expectancy and mortality rates. In the mid-nineteenth century, actuaries became an established profession with accreditation requirements, earning the title of actuarial science.

In the 1970s and 1980s, financial institutions began focusing on risks that were not insurable, such as credit risk, market risk, and liquidity risk. In the 1990s, after a series of corporate failures resulting in significant losses, the UK Financial Reporting Council (FRC) established a commission that led to the Cadbury Report (1992) [14]. Subsequently, the London Stock Exchange and the Institute of Directors (IoD) set up another commission that resulted in the Turnbull Report (1998) [15]. These reports identified recommendations for corporate governance and internal control systems, influencing changes in the UK Corporate Governance Code and listing requirements. Executives

now needed a holistic view of the risk landscape, leading to a change in approach toward qualitative scoring and the concept of Enterprise Risk Management (ERM). In 2004, the Committee of Sponsoring Organizations of the Treadway Commission (COSO) released its Enterprise Risk Management—Integrated Framework, providing a comprehensive structure for implementing ERM within organizations.

It has been over 20 years since Enterprise Risk Management became a framework and methodology. It is remarkable to see the widespread adoption of formal ERM programs, particularly among large organizations. ERM has become a substantial industry with significant resources dedicated to services, software, and efforts within organizations.

While the evolution of this concept has been remarkable, there are still opportunities for further development. According to an EY Global Board Risk Survey, 84% of boards do not believe their organizations have a highly effective risk management strategy [16]. As consistent with the theme of this book, the ERM profession will be influenced by the same technological enhancements and innovations driving the digital transformation movement.

Having spent a significant number of years consulting in the banking industry, advising clients and executives on strategies to address their major risk management challenges, I have identified the biggest opportunities, supported by other industry research I will reference.

Moving Beyond Qualitative Risk Assessments

In his book *The Failure of Risk Management: Why It's Broken and How to Fix It* [13], author Douglas Hubbard highlights the shortcomings of the qualification approach to measuring risk.

According to Hubbard, many risk management surveys, including the EY Global Board Risk Survey mentioned earlier, often rely on questions like “How do you rate your risk management strategy?” The problem with such surveys is that the responses are subjective opinions that lack empirical evidence to support them. This raises concerns for senior leaders and boards as they question whether their risk management programs truly have a grasp on all major and emerging risks. There is a prevailing perception that ERM is effective in addressing past issues but struggles to proactively address emerging risks.

THE 2008 FINANCIAL CRISIS

The 2008 financial crisis serves as a prime example of a failure that can be attributed to a lack of quantification. The root cause of the crisis was the relaxation of underwriting standards, with approximately one-third of all mortgages in 2006 being subprime or no documentation loans, accounting for 17% of home purchases that year [17].

Numerous actors played a role in the crisis, including government-sponsored entities like Fannie Mae and Freddie Mac, regulators, mortgage lenders, credit rating agencies, rating institutions, and financial institutions. Any of these actors could have assessed the sensitivity of those lower-quality mortgages to a potential housing bubble, but none of them did so or took appropriate action, potentially due to their self-interest. The resulting failure had catastrophic consequences for various stakeholders, including the global economy, homeowners, taxpayers, investors, and others.

The basis for assessing risk is the risk matrix. Figure 7-2 provides an example of a risk matrix, where risk assessors plot the likelihood and impact of a risk. In this example, each axis of the matrix is assigned a value from 1 to 5. An aggregate score is calculated by multiplying the likelihood and impact, resulting in a risk designation ranging from low to high. This process helps translate a qualitative model into a semiquantitative model, facilitating the prioritization of different risks for mitigation.

Superimposed on the risk matrix are three distinct risk level designations commonly used to evaluate control effectiveness. Inherent risk refers to the risk level before implementing any mitigation measures. Residual risk represents the risk level considering the current controls in place. Target risk represents the desired risk level to be achieved through further mitigation efforts.

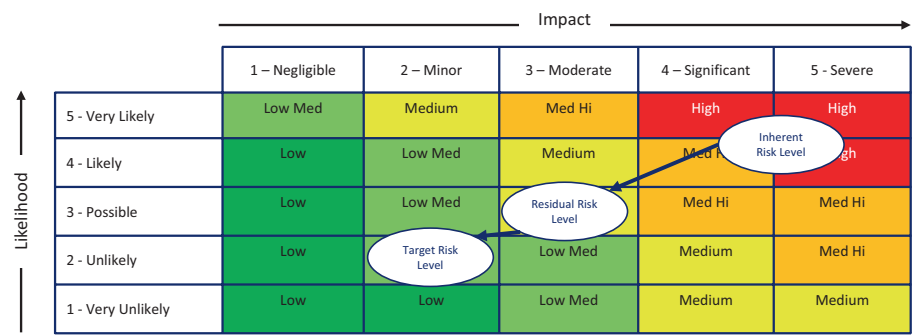


Figure 7-2. Risk Matrix

Given the need for Enterprise Risk Management (ERM) to provide a comprehensive summary of the risk landscape for executive consumption and the reality that it is not feasible to create a quantifiable model for every risk, it is unlikely that the risk matrix will be eliminated anytime soon. However, to identify opportunities for improving ERM, we must have an honest understanding of its limitations.

The current method of assessing risk is a subjective process carried out by humans, who are prone to various biases. Here are some of the biases that may impact scoring:

- **Confirmation Bias:** This refers to our tendency to interpret the world in a way that confirms our existing beliefs or values. An example of this is the 2008 financial crisis, where individuals within different actor groups could have recognized the impending disaster, but their cognitive biases only allowed them to see what aligned with their financial success.
- **Interpretation Bias:** This bias involves interpreting information or situations in a way that is influenced by preexisting beliefs, expectations, or emotions. Since the assignment of likelihood and impact is based on subjective judgment, scoring may have been interpreted differently across organizational boundaries, hindering consistency, prioritization, and decision-making.
- **Groupthink Bias:** Groupthink occurs when a group of people, aiming for consensus and cohesion, prioritize agreement and harmony over critical thinking and independent evaluation of ideas. This can lead to overconfidence and downplay certain risks while suppressing important dissenting points of view.

By acknowledging and addressing these biases, organizations can strive for a more objective and robust risk assessment process within their ERM frameworks.

Midsized and large corporations are typically complex systems in which various parts of the organization collaborate to serve customers or fulfill important purposes. Often, assessments are conducted within business or functional units without a comprehensive consideration of upstream or downstream processes. When assessments are performed across organizational boundaries, a lack of common language and understanding of individual processes hinders the ability to grasp their interdependencies. This challenge is particularly pronounced for teams lacking functional expertise, such as internal audit. Consequently, risks that are deemed low at the component level may result

in high failure when combined. Analyzing the correlation of events along the value chain can provide valuable insights and impact risk ratings along the end-to-end process.

Similarly, there is the challenge of risk aggregation. While scoring in the risk matrix is performed for individual risks, it may fail to capture the compounding effect in which a certain number of medium risks can lead to a high-risk situation. Ultimately, when presenting a report for executive consumption, it is crucial that the view represents an accurate depiction of overall risk.

The lack of granularity in process and contextual information, including the definition of processes, process boundaries, and historical negative events such as risks and operational losses, can contribute to an incomplete identification of potential failures.

The opportunity lies in defining and implementing strategies that instill confidence in senior leaders, providing them with an accurate assessment of the risk landscape and enabling better-informed strategic decisions. This entails quantifying risk to a greater extent and equipping risk assessment teams with more granular information and robust methods to enhance accuracy and mitigate human bias.

Need to Improve Risk Data

Unfortunately, poor data quality is prevalent in most large organizations. According to the Forbes Insights and KPMG study [18], “84% of CEOs are concerned about the quality of the data they’re basing their decisions on.” This statement is concerning as it not only affects the ability to effectively manage risks such as operational losses and regulatory fines, but it also hinders the organization’s risk manager from having a seat at the table in strategic decision-making.

CITIBANK CONSENT ORDER

In August 2020, Citibank intended to make an interest payment to lenders on behalf of Revlon, a company for which Citibank served as the loan agent. However, due to a combination of human error and a lack of adequate safeguards in the payment system, the full principal amount of the loans, totaling nearly \$900 million, was transferred to the lenders.

As a fallout of this incident, the US Office of the Comptroller of the Currency (OCC) issued a Consent Order [19]. A Consent Order is essentially a directive from a regulatory body to a company to address its compliance shortcomings, which in this case included deficiencies in Citibank’s risk management practices and internal

controls, particularly in operational risk. A significant aspect of the Consent Order was related to data issues, particularly in relation to risk data, which prevented Citibank from identifying this risk and implementing mitigating controls. Article II of the Consent Order specifically addressed concerns regarding data quality issues.

The OCC has identified the following deficiencies, noncompliance with 12 C.F.R. Part 30, Appendix D, or unsafe or unsound practices with respect to the Bank's data quality and data governance, including risk data aggregation and management and regulatory reporting:

(b) inability to develop and execute on a comprehensive plan to address data governance deficiencies, including data quality errors and failure to produce timely and accurate management and regulatory reporting; and

(c) inadequate reporting to the Board on the status of data quality and progress in remediating identified deficiencies.

This Citibank example is a great illustration of a **nonfinancial risk** failure. Nonfinancial risks are the risks that occur in the normal course of doing business. To safeguard against these risks, it's imperative to have a detailed understanding of all the business-as-usual activities so that an accurate assessment of risk can occur and effective controls can be implemented.

In the case mentioned earlier, Citibank did not have controls in place to guard against such a massive errant payment. It is fair to say that their risk assessment process was flawed as this payment process should have scored high in both its inherent and residual risk ratings, considering its capacity for such a massive payment. This would have made it a high priority, demanding focused efforts to implement stronger mitigating controls. The root cause of this failure seems to be that their risk management program literally lost sight of the existence of this processes. This is not uncommon in an enterprise as large and complex as Citibank, where there are hundreds or thousands of processes that can pose similar risks. That is why it is imperative to build and maintain high-quality risk data so that the highest risks are addressed with the necessary urgency. According to the text of the Consent Order, it appears that poor data quality was at the heart of this issue.

In a recent Deloitte Global Risk Management Survey, it was reported that “most respondents said their institutions found certain issues to be extremely or very challenging: maintaining reliable data to quantify nonfinancial risk and drive risk-based decisions (74%) and the ability to leverage and source alternative data, such as unstructured data (74%)” [20]. Only about one-quarter of respondents believed their institutions are extremely or very effective at managing data quality (26%) for nonfinancial risks. Only about one-quarter of respondents believed their institutions are extremely or very effective at managing data quality (26%) for nonfinancial risks. This means that this issue is pervasive across institutions.

Since an organization can have a significant quantity of risks, risk stakeholders, and the need for accurate risk reporting, organizations must create and maintain a repository of information from which to manage and report on the risk landscape.

For organizations that are maturing their risk management capability, this may mean a risk registry spreadsheet that is built and maintained in each organizational unit. The challenges with this are obvious: spreadsheets are prone to quality issues. Additionally, aggregating information across the enterprise for senior-level decision-making would be a time-intensive activity to reconcile each unique structure into a coherent story.

A common tool for mature ERM programs is a governance, risk, and compliance (GRC) platform. GRC tools encompass the systems, processes, and practices that enable an organization to achieve its risk objectives while effectively managing risks and complying with applicable laws, regulations, and internal policies. GRC tools can be built in-house, but there are also vendor software platforms. The global vendor market is expected to reach \$134 billion by 2030, with a compound annual growth rate (CAGR) of 13.8% [21]. Adoption is continuing at a healthy pace.

Here are the capabilities of these platforms that will aid an ERM program:

Governance refers to the capabilities of leadership to provide oversight and control of the ERM program to ensure it aligns with business goals. These platforms enable management to document and manage policies and procedures and track compliance from the stakeholders involved. Stronger decision-making is promoted by aggregating risk information through reporting into actionable items for senior leadership and board consumption.

Through that, it connects the dots between strategic decision-makers and risk practitioners across the organization by facilitating accountability and stronger coordination across internal stakeholders since all teams would work off the single source of truth. Integration with external stakeholders, such as external auditors and regulators, would benefit from reports generated from this single source.

Risk capabilities provide a structured approach to managing risk. They support all steps in the risk management process, such as identification, assessment, prioritization, and documentation of mitigating controls. They provide monitoring and reporting capabilities by aggregating data from multiple sources. The platform can model scenarios so risk practitioners can assess the impact across these scenarios. It promotes the risk culture by documenting ownership of the various risk concepts. Workflow capabilities track activities in risk processes, send out reminder emails, track responses, and provide an audit trail.

Compliance capabilities help organizations comply with regulatory requirements, standards, and industry guidelines by providing a repository to map regulations, the resulting obligations for the organization, and the policies, procedures, and controls put in place. These platforms can monitor changes in regulations so that changes can be implemented. Compliance reporting and dashboards facilitate productive conversations with regulators.

A foundational feature of GRC platforms is providing a centralized repository for risk information. I've provided a partial and illustrative GRC data model in Figure 7-3. As you can see from the diagram, this is more than just a registry of risk; it also stores many other data subjects, such as processes, controls, regulations, and monitors. Each data subject may have distinct stakeholders that provide input and leverage this data in their roles. It's critical that this data and the associations across the concepts are governed to maintain their fidelity.

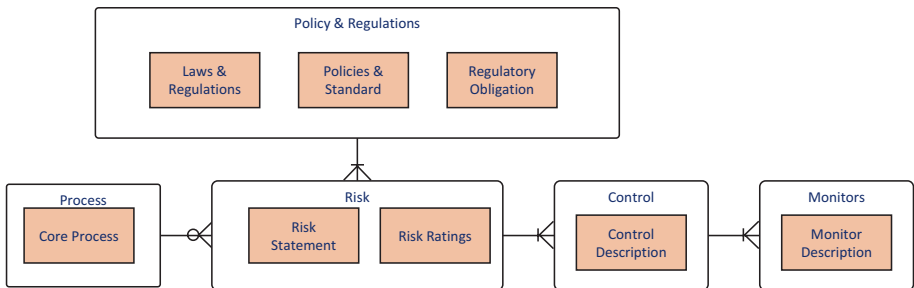


Figure 7-3. Illustrative GRC Data Model

A process can have zero to many risks. Every risk must be associated with a process as internal risks do not occur without some action trigger, which is represented by a process. This association also provides business context and accountability if process owners have been designated. Every risk has one or more controls, and each control should have one or more monitors. I will cover regulatory obligations a bit later in this chapter.

In my experience, a prominent root cause of nonfinancial risk failures is poorly managed data. Since the activities that an organization performs are the source of risk, I cannot stress enough the importance of having an accurate inventory of those activities, specifically processes, as a key component of this data structure.

Unfortunately, I have yet to come across an organization that has mastered this aspect. Much of the standard ERM literature refers to aligning with “core processes,” but this term is vague and subject to different perceptions among stakeholders. Other organizations may adopt a standard process framework such as APQC’s Process Classification Framework. However, the challenge

with this approach, especially within organizations with disparate business units, is that these standard models rarely reflect the granularity of business activities due to their unique idiosyncrasies. As a result, risk assessors and business stakeholders must interpret the activities performed by the business and map them to the standard model for storage, which often leads to data quality issues.

There are new GRC platforms emerging called Cognitive GRC, which integrate traditional GRC platforms with the capabilities of artificial intelligence and machine learning. This integration significantly enhances the capabilities of these platforms.

From a GRC perspective, I still believe that high-quality data is the key to an effective ERM program. Prior to undertaking a GRC implementation, it is critical to carefully consider the ontology of risk concepts and identify the appropriate data model for your organization.

2012 JPMORGAN CHASE TRADING LOSS

In April and May of 2012, JPMorgan Chase's Synthetic Credit Portfolio (SCP) group, which traded credit default swaps (CDSs), suffered substantial losses, initially estimated at \$2 billion but later revised to \$6.2 billion [22]. This incident represented one of the largest losses in financial history.

There were several control breakdowns that contributed to this incident. The SCP relied on financial models built into spreadsheets that assumed a correlation between short-term and long-term hedges, which had become invalid. Value-at-risk (VAR) limits were breached over 300 times, and traders were given the freedom to choose their preferred daily intraday price range, effectively marking their own positions.

However, one significant failure was the lack of comprehensive position reports on the portfolio provided to the chief investment officer, who was responsible for overseeing the SCP. As a result, senior executives lacked a complete understanding of the risks until it was too late.

The preceding JPMorgan Chase example highlights a failure in **financial risk** management. Financial risk refers to the possibility of losing money in an investment or business venture. This category encompasses

- **Credit Risk:** The risk of financial loss resulting from counterparties failing to meet their obligations, such as defaulting on loans or credit.
- **Market Risk:** The risk of financial loss resulting from adverse events in market prices, such as interest rates, exchange rates, commodity prices, or stock prices.

- **Liquidity Risk:** The risk of not being able to meet financial obligations as they become due because of insufficient liquid assets or the inability to quickly sell assets without significant loss. Liquidity risk can arise from cash flow mismatches, reduced marketability of assets, or disruptions in funding sources.

Assessing and quantifying financial risk involves building models to quantify financial positions. This is also true for some regulatory compliance reports such as capital adequacy reports. These models require high-quality, accurate, granular data with auditable data lineage. This extends beyond GRC as the data needs to be sourced from multiple transactions, asset portfolios, or finance systems, emphasizing the need for data quality beyond just risk purposes.

Effective financial risk management means that senior leaders, and in some cases regulators, trust the reports guiding their decision-making in mitigating risks. There is ample opportunity through data governance and data architecture to increase confidence in risk management data, providing senior leaders with assurance that their decisions are based on solid factual foundations.

Risk Operating Model Opportunities

A standard model for identifying accountabilities and delineating responsibilities within an Enterprise Risk Management program is the Three Lines of Defense (3LoD) model. The exact origin of this model is unclear; its roots seem to be in the banking industry, with the Basel Committee on Banking Supervision (BCBS) publishing a position paper titled “Sound Practices for the Management and Supervision of Operational Risk” [23] in 2003. Although the paper did not explicitly name the 3LoD model, it outlined many principles for effective risk management and supervision that have since become associated with the model. The 3LoD model has expanded beyond banking and is now commonly practiced in many industries, especially after the Institute of Internal Auditors (IIA) formally adopted the model in 2013. However, the level of adoption varies based on the level of regulatory scrutiny as it is a model that regulators expect.

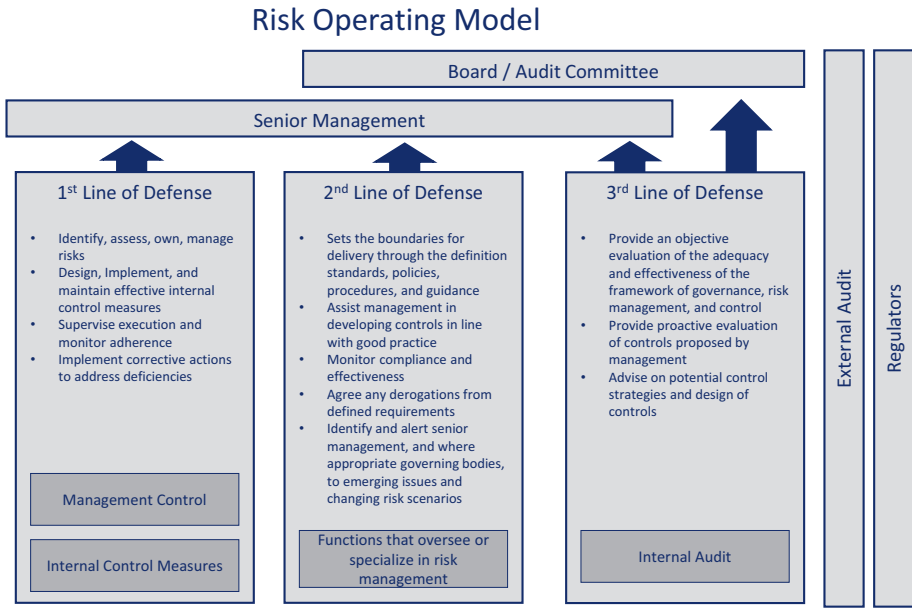


Figure 7-4. Three Lines of Defense Model

The 3LoD model is highlighted in Figure 7-4, which was sourced from The Orange Book Management of Risk – Principles and Concepts [24]. The roles of each line of defense are as follows:

The **first line of defense** is embedded within the business units or supporting functions (such as technology, finance, etc.). The business units have direct responsibility for delivering products and services to clients while supporting functions directly support the goals of the business unit or enterprise. This group possesses in-depth knowledge of their processes and takes the lead in risk assessments and control designs. The first line executes controls in their business unit or function which are not technology-based, and they are accountable for the performance of those controls. Control steps, for manual controls, are typically outlined in process procedure manuals for staff.

The **second line of defense** is responsible for monitoring and overseeing risks. In many organizations, the second line is organized around the risk taxonomy, such as operational risk, compliance risk, cybersecurity risk, etc. They bring domain expertise and establish policies, procedures, and standards that guide the first line in the risk process. They monitor and review the work of the first line to gain assurance of the effectiveness of controls. They also prepare reports and dashboards for management and board review, providing an aggregated view of the risk landscape across business units and functions.

In some smaller organizations or those with lower regulatory requirements, the first and second lines may be combined in the organizational structure.

The **third line of defense** is the independent internal audit function. They perform objective evaluations to assess the adequacy and effectiveness of internal controls and provide recommendations on improving risk management and governance. They provide assurance directly to senior leadership and the board that the risk management processes are operating effectively. While they can provide feedback and advice to the first and second lines of defense, they cannot design controls, which is essential to maintaining their independence.

Although the Three Lines of Defense model has been around for over 20 years and is pervasive in many industries, significant challenges exist in its implementation. In a recent Deloitte Global Risk Management [25] Survey of 57 financial service institutions worldwide, the following concerns were highlighted:

The most frequently cited challenges concerned the responsibilities and capabilities of the first line, including

- 44% gaining buy-in from the first line
- 50% defining the roles and responsibilities between the first line and the second line
- 33% executing first-line responsibilities
- 39% having a sufficient number of skilled personnel in the first line

These results are not shocking as they align with my experiences in various financial services organizations. I also assume this is not limited just to financial services and that other nonfinancial service organizations are experiencing similar challenges.

The opportunities to improve this model lie in defining and adopting stronger leadership and governance, implementing a stronger accountability model, and enhancing collaboration among the three lines.

Leadership for each line of defense can be strengthened by improving how they govern their teams. This involves cascading goals down the organization and enhancing risk management reporting with more quantifiable metrics to drive a culture of continuous improvement.

To enhance accountability, it is crucial to provide clarity on individual accountabilities. These accountabilities should be formalized in yearly performance goals, and performance based on these accountabilities should be integrated into the annual review process. While documenting accountabilities is important, measuring individuals' performance and aligning financial incentives can further reinforce the desired behaviors.

These accountabilities should also be reflected in the risk data mentioned earlier. I worked with a marketing team at a Global Financial Services organization, and I observed that their risk data did not even include the name of the accountable first-line business owner for each risk and control. Without such clarity, things are bound to fall through the cracks as no one takes ownership of individual controls.

Lastly, it is essential to establish clear areas of collaboration between the lines. Tensions are inevitable between the lines as the first line is often focused on revenue generation and business results, while the second line tends to be risk-averse. Recognizing potential points of conflict and fostering collaboration between teams, with the involvement of their leaders, are crucial for negotiating solutions that meet the needs of the business. This is the path to breaking down silos in the ERM program.

The Future of ERM Fueled by Process Inventory

The first part of this chapter was dedicated to the history of Enterprise Risk Management and the challenges faced by those programs. The remaining part of this chapter will focus on a vision for the future of ERM, fueled by the Process Inventory framework. It will include a summary of that vision as well as some practical examples of how the future of ERM would work.

There are several trends influencing the future of ERM, many of which are driving the overall digital transformation. Here are some of the most pressing drivers that I would like to highlight.

Technology poses the biggest disruption while also offering opportunities for the future. To illustrate this disruption, let's consider regulatory compliance. Previously, legal and compliance teams had to engage in manual and intensive efforts to understand changes in laws and regulations. Now, machine learning can accomplish this task in a fraction of that time with a high degree of accuracy. This productivity multiplier allows resources to focus on other high-value tasks. Furthermore, machine learning can continuously process internal and external signals to validate existing controls and detect patterns that point to new threats. However, this opportunity is only available to organizations that recognize **data as an asset**. The proliferation of data in recent years necessitates a data management approach to organize and ensure the quality of the data, enabling the extraction of valuable insights for better business decisions.

We are currently experiencing an era of significant change, where **competition** from disruptors could pose a threat to established players, particularly in highly regulated industries where regulatory requirements act as entry barriers. Innovation, not only in risk management but across all aspects, is increasingly critical for organizations that want to survive and thrive.

The **regulatory** landscape is becoming increasingly complex. The winners of the future will establish strong operating models with clear processes to ensure that compliance doesn't create unnecessary complexity within their organizations. Lastly, globalization and the risk of interconnected global **supply chains** will persist. Organizations must not only understand their own risks but also have mitigation measures in place in case of a failure by upstream or downstream partners.

Here are three strategies to succeed in the future:

Strategy I—Reconstructing the Risk and Control Framework

The current state of complexity in risk data needs to adapt not only to address the challenges mentioned earlier but also to become agile in the face of a continually changing landscape. This cannot be emphasized strongly enough: without accurate risk data, the complexity of the operating model will persist, and the quality of information on which leaders base their decisions will always be questioned.

This means orienting the framework toward processes. Logically speaking, an internal risk cannot be incurred without some trigger taking place. For internal triggers, that means an action, or in other words, a process. Anchoring the framework to a single truth for processes, Process Inventory is foundational to getting the business context right and complete, thus ensuring high-quality risk data.

Aligning with processes brings **clarity and alignment** to the risk management environment. The root of the challenges is not the complexity of the business itself, but rather the different levels of understanding across the practitioner teams. These disparate levels of understanding result in gaps in interpretations, which cause inefficiencies and errors. This approach effectively breaks down silos as everyone will have a common and detailed language to work from.

This alignment results in **faster root cause analysis** of issues as the resources that support the process, such as technology systems, are directly aligned. Getting technology resources on the same page of understanding can facilitate the shift from manual and reactive controls to automated and preventative controls, which are less susceptible to human error. This has the side benefit of freeing up resources for higher value activities.

With the Process Inventory providing an accurate and comprehensive inventory of all processes, it gives risk teams a denominator from which to evaluate the **completeness** of their efforts. The outcome would provide a higher confidence in the periodic assessment of risk, as each process area would have to be reviewed, or a conscious decision made not to review an area. Regardless, nothing will fall through the cracks.

This structure empowers risk teams to extend risk ratings beyond the risk itself and aggregate a measure for rating the riskiness of processes. This puts it in a language that business leaders understand and gives them insights into which processes require investment to strengthen.

This structure becomes the anchor point within the GRC data structure, mapping all the data concepts, such as regulatory obligations, business continuity events, etc., to the processes impacted. As Figure 7-5 illustrates, this model brings frontline execution teams as process owners, ensuring **higher data quality**, as they can validate the accuracy of process information.

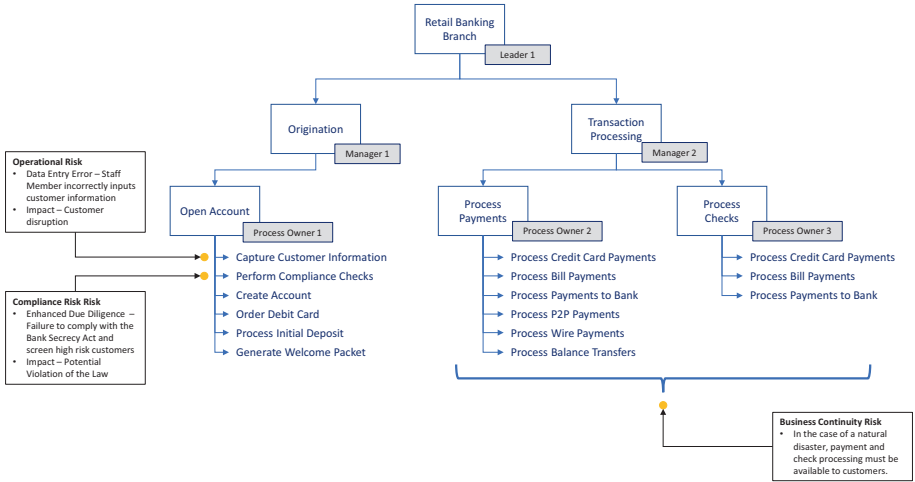


Figure 7-5. Process Inventory Risk Alignment Example

Strategy 2—Strengthen the Risk Management Operating Model

Accountability is the key to driving a high-performance culture. Many ERM programs place a significant emphasis on culture, which is critical, but it must be accompanied by setting clear expectations, establishing goals that cascade throughout the organization, and holding everyone accountable to those goals.

As illustrated in Figure 7-5, each process in the Process Inventory is designated with ownership throughout the structure. Process owners, who are part of the first line of defense, are key since they attest to the accuracy of the inventory and have the best understanding of the process details. Process owners must play an active role in the risk assessment process and sign off on the controls for which they have accountability. Since the Process Inventory aligns with the organizational structure, a control for which a process owner is responsible is also a measurement for their leader and that leader’s leader. This means accountability is cascaded at all levels of the organization. Through this model, risk management is perceived not as a burden that distracts from their day job but as a core part of their responsibilities.

There are many second-line defense functions that currently operate in silos and compete for the limited attention of first-line leaders and process owners. As depicted in Figure 7-5, processes in the inventory can have risks of almost any type. Aligning them to a single source of process truth will break down silos and foster better coordination through the process lens. Second-line functions will be accountable for reviewing the accuracy of risk assessments and ensuring that controls align with their policies. This overlap of goals will lead to better collaboration.

For the third line of defense, this approach should make their job much easier. In my experience, internal audit teams have struggled to understand how the business operates, and I've seen many of them establish process teams in an attempt to gain clarity. Since this structure provides them with clarity on the business's operations, they can focus their energies on testing controls and evaluating the overall control environment. Having near real-time data on control performance will assist them in identifying areas of focus when they construct their audit plan.

Strategy 3—Exploit Technology Advancements to Achieve a New Level of Risk Intelligence

Machine learning can analyze processing events to identify patterns, make predictions, and provide risk managers with insights into the strength of existing controls, as well as alert them to emerging risks. To capitalize on this potential, machine learning will have to be given labeled data [26] to understand the boundaries between these events and to correlate data from disparate sources. Training machine learning with labeled data powers data-centric artificial intelligence and facilitates effective supervised learning.

The Process Inventory is the structure for demarcating processing events and provides the perfect structure for labeling data. The data sets for machine learning to learn from include transactional data, control performance data, negative event data (such as issues, complaints, operational losses, etc.), system data, and performance data. To fully leverage the potential of these sources, organizations must align their data with a consistent data structure, which is provided by Process Inventory.

Machine learning technologies will also be useful for horizon scanning of the external environment to identify emerging trends, events, and potential risks that may impact an organization. These include economic trends, regulatory and compliance changes, geopolitical risks, natural environmental risks, and even social media trends that may point to emerging risks. Many organizations already perform horizon scanning, but with machine learning technologies, the efficiency and level of insights will result in stronger response plans.

This would provide leadership with a dashboard framed in a business context to understand the health of the control environment. This is demonstrated in Figure 7-6; this example dashboard would give leadership and other risk

professionals the ability to drill down into the performance of any individual process area or process. Moreover, it would provide machine learning with a rich set of quality data to identify emerging patterns and mitigate human bias in the risk assessment process.

Machine learning provides the ability to analyze data and run mathematical models, which can move the needle from qualitative measurement of risk to quantitative. For instance, the Monte Carlo simulation incorporates input variables, each with a range of possible values or probability distributions based on available data or assumptions. It then runs simulations using the distribution of these variables to provide a range of possible outcomes with their probabilities. This approach applies not just to a single process or risk but to the larger ecosystem of interconnected processes and risks, uncovering risks that may otherwise go unidentified using traditional methods. This provides teams with richer insights to identify and design stronger controls.

The result of these enhancements is that senior leaders will have assurance that they are getting an accurate picture of the risk landscape to aid them in strategic decision-making. Near real-time dashboards will provide all leaders with the state of the control environment in the appropriate business context, which is viewed through the lens of the Process Inventory. All of this would be achieved while driving better performance in risk processes and being efficient with resources.

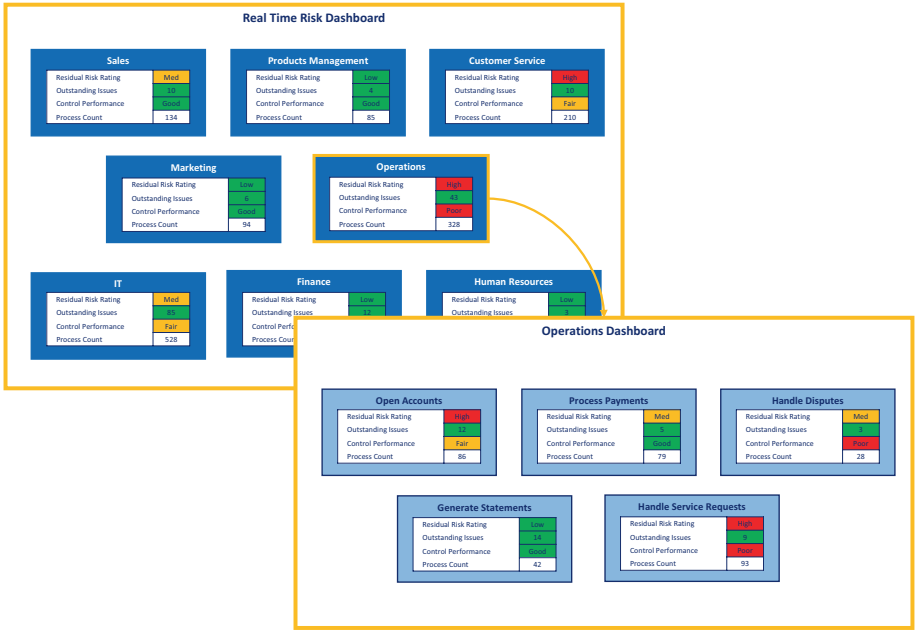


Figure 7-6. Process Aligned Real-Time Risk Dashboard

Defining the Enterprise Risk Management Framework



Figure 7-7. COSO Enterprise Risk Management Framework—Integrating with Strategy and Performance

Figure 7-7 represents *COSO’s Enterprise Risk Management—Integrating with Strategy and Performance Framework*, which provides guidance to the board and management in defining risk strategy.

As mentioned earlier, the US Securities and Exchange Commission (SEC) mandates that boards of publicly traded companies ensure risk oversight, indicating that the tone for risk management begins at the top. Boards may establish dedicated risk committees or assign primary risk oversight to the audit committee. According to the Deloitte Global Risk Management Survey, regulators increasingly expect risk committees to include independent directors with risk management expertise, and this expectation has influenced board compositions. Seventy percent of respondents stated that their board’s risk committee comprises either entirely or mostly independent directors [25].

Boards have several oversight responsibilities. They review the organization’s formal risk governance framework, which outlines the structure, processes, and responsibilities of risk management within the organization. They also review the risk policy, which outlines the organization’s approach to identifying, assessing, and managing risk. Boards are involved in defining the enterprise-level risk appetite statement, which represents the level of risk that the organization is willing to accept in pursuit of its business objectives. Additionally, they regularly review risk management reports and provide guidance, leveraging their expertise, on the top risks facing the organization.

The management of risks falls under the responsibility of the organization’s leadership, starting with the CEO. CEOs often appoint a senior risk manager, which may be designated with the title of chief risk officer, to oversee the risk management program. In many cases, the risk managers report directly to the CEO and provide reports to the board.

The primary role of the risk manager is to establish the framework, processes, and policies of the Enterprise Risk Management (ERM) program. Increasingly, risk managers are involved in strategic decision-making, requiring them to possess not only risk management skills but also a deep understanding of the business. This enables them to integrate a risk perspective into strategic decision-making processes.

The CEO and the risk manager must work together to foster a risk-aware culture throughout the organization. This culture begins with accountability. They need to define a clear operating structure with well-defined accountabilities and goals that cascade throughout the organization. It should be evident to everyone how risk management contributes to their performance evaluations. The culture must strike a balance between risk management, innovation, and business growth.

Risk managers take the lead in risk governance efforts and are responsible for producing comprehensive reports for the CEO and the board. These reports provide a holistic and accurate summary of the risk landscape, highlighting key risks that require attention and guidance from the leadership.

The Process Inventory framework is a valuable tool for these leaders. It offers a holistic view of processes and provides a structured approach to aggregate and quantify risks, ensuring a comprehensive understanding of the risk landscape. The framework strengthens the operating model by establishing a hierarchy of processes with clear ownership, enabling the cascading of goals throughout the organization, and reporting the status of controls.

Ignite ERM of the Future with Data Excellence

Data has presented challenges for organizations, but when managed properly, it can serve as a robust foundation for enhancing the effectiveness of risk management programs. In this section, we will explore how the Process Inventory can contribute to high-quality GRC risk data and how it provides a framework for unlocking valuable insights. By harnessing the potential of the Process Inventory, organizations can elevate their risk management practices and leverage the transformative power of data-driven decision-making.

High-Quality GRC Risk Data

Risk data, stored within a GRC repository, serves as the critical foundation for effective risk management practices. This data plays a pivotal role in facilitating informed risk identification during the risk assessment process. Moreover, it forms the basis for comprehensive reporting on the organization's risk landscape, enabling strategic decision-making based on accurate and

timely insights. Additionally, risk data serves as tangible evidence of the organization’s compliance with relevant laws and regulations, providing assurance to regulators.

Since Process Inventory is constructed by following the organizational hierarchy and each point in the Process Inventory taxonomy has an identified owner, it fosters a culture of accountability and strengthens the organization’s risk management framework. Identified process owners become the core of the first line of defense. Data quality is achieved by periodically requiring process owners to attest to not only the processes in their inventory but also the associated risk data.

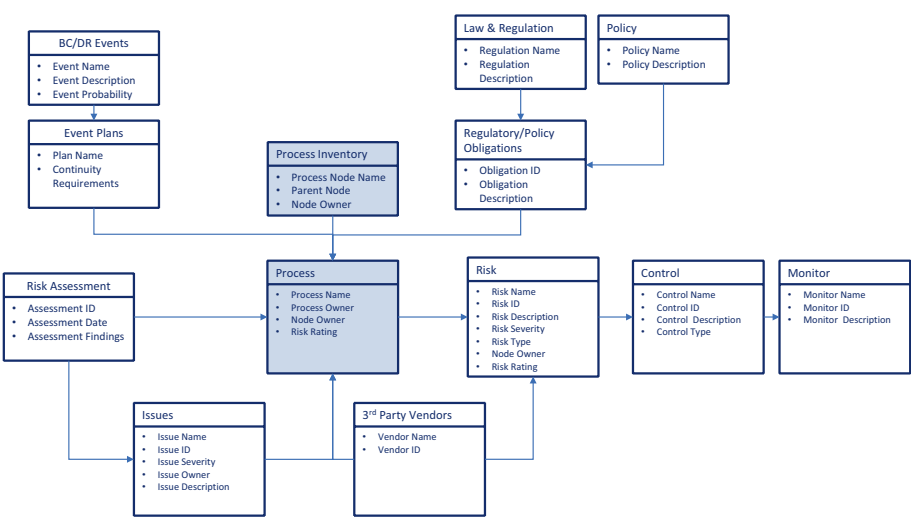


Figure 7-8. Illustrative Target State GRC Data Model

Figure 7-8 illustrates the GRC data model with Process Inventory as the grounding truth for business context. Process Inventory and the processes are highlighted in blue. These data would be built and maintained in the Integrated Operational Repository. This data would be exported to the GRC repository. Since process owners would be active participants in the risk assessment process, they would be required to attest to the accuracy of the process information for which they are an owner as part of the assessment process. This would ensure that assessments are performed on accurate process information.

Here are some of the relationships that are key in this model:

- **Risk** is associated with a process, as risk must be triggered by an activity, which for internal risks is a process. Each risk can have one or more controls, and each control can have one or more monitors.
- **Laws, regulations, and policies** have obligations which are the interpretation of those rules and the requirements that it places on processes. Those obligations need to be placed on processes so process owners can design the appropriate controls needed to comply.
- **Third-party vendors** can provide a wide range of services to an organization such as being a business process outsourcer, providing software platforms, and providing professional services. For business process outsourcers, it's important to remember that even if the process is outsourced to a vendor, the accountability remains with the organization. Due to that, Process Inventory should contain an inventory of processes that the vendor is performing so that risk management teams can have clarity on which controls that vendor is required to implement. For software platforms, it's important to know which processes are served by those platforms for root cause and monitoring purposes.
- **Issues** are associated with the processes that incur them. This relationship provides the triage and resolutions team with access to not only the controls associated with those processes but also all the resources that are leveraged.
- **Risk assessments** are performed on processes so that relationship must be maintained in this data model. Findings are placed on individual processes which place accountability for the resolution on the process owner.
- **Business continuity/disaster recovery events** are predominantly external events that can cause disruption to operations. Continuity plans are created for each scenario which includes which processes must be maintained, even at a degraded level. This also enables the design team to look for points of weakness in processes, such as a single data center in a hurricane zone, and remediate those weaknesses before the event occurs.

Data-Driven Risk Insights

Risk intelligence is a term that has been around since the 1980s. David Apgar defines it in his book *Risk Intelligence: Learning to Manage What We Don't Know* as “experience, any and all experience, past and future, that can help us solve problems requiring an understanding of risk” [27].

Organizations generate a massive amount of data related to the experiences of their business processes. This data holds valuable insights into better performance, especially in terms of risk management. However, most organizations lack a structured approach to extracting these insights and applying them to the risk management process.

The core challenge is that we are missing a single language for correlating data across these disparate platforms and for demarcating process signals. With the power of machine learning, addressing this challenge has become more urgent. Machine learning technology enables us to process information, detect patterns, and test our current risk assumptions, helping us identify weaknesses that were previously unknown.

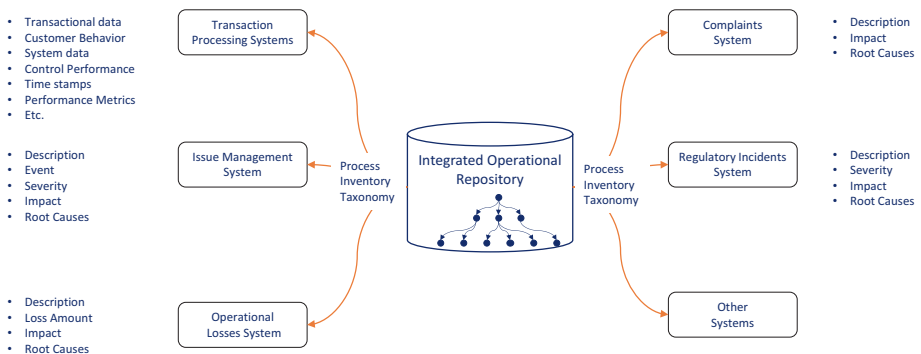


Figure 7-9. Publishing Process Inventory Taxonomy

The Process Inventory serves as a unified language for consistently categorizing data in various systems with the appropriate business context. Numerous systems generate and manage data on process performance, encompassing both positive and negative aspects. The key opportunity lies in establishing a common language among these systems.

Figure 7-9 depicts how the Process Inventory taxonomy is disseminated from the Integrated Operating Repository to these systems. Transaction processing systems would need to align their data with this standardized language through coding or configuration. On the other hand, systems like issue tracking or

operational loss systems, which rely more on manual processes, would benefit from the Process Inventory's classification system. It allows triage teams to categorize each record based on the process that generated the incident.

The data derived from these systems, combined with the insights generated by machine learning, would enhance the work of risk management teams during risk assessments. It strengthens their risk inventory and aids in designing effective controls. Moreover, this data serves as input for mathematical models that quantify risk, enhancing the precision of their output.

Risk Assessment Examples

According to a Deloitte article, "Organizations can struggle with how they consolidate multiple risk assessments and reduce the duplication of efforts across assurance functions, which can lead to gaps in risk coverage and inconsistent approaches to managing risks" [28]. This underscores some of the previously mentioned challenges in the risk operating model between the first and second lines of defense.

In this section, I will demonstrate how risk assessment across different risk types in the risk taxonomy can be coordinated through the lens of process and how this will lead to much better coordination and clarity in roles and responsibilities.

Risk Taxonomy

A robust ERM program relies on a well-developed risk taxonomy as its foundation. A risk taxonomy establishes a standardized language for identifying, assessing, and managing the diverse range of risks faced by an organization across its business units and functions. It takes the form of a hierarchical structure, with higher levels encompassing broader risk categories and lower levels offering more precise definitions. By utilizing a risk taxonomy, organizations can benchmark risks, detect trends, ensure consistent aggregation and reporting, define risk appetite, and effectively allocate resources. The risk taxonomy has implications for the risk operating model as it provides the structure for how an organization manages risks, which often defines the framework of the second line of defense.

While several risk standards organizations provide a basic framework for classifying risks, an effective risk taxonomy must be comprehensive and tailored to an organization's unique risk landscape. Industry-specific reference risk taxonomies can serve as valuable starting points as they address common risks arising from shared markets, regulatory requirements, and operational complexities. However, organizations must customize and refine the risk taxonomy to align with their specific circumstances, accurately reflecting the risks they encounter.

By employing a well-designed and comprehensive risk taxonomy, organizations can enhance their risk management practices, promote consistent understanding and communication of risks, and make informed decisions to mitigate and address potential threats.

Integrated Risk Assessment

To demonstrate the benefits of Process Inventory, let's focus on an integrated risk assessment process. This approach involves assessing all risk types that impact a business unit or function in a single assessment, providing a comprehensive view of risks and enabling better prioritization and resource allocation. This stands in contrast to organizations that assess different risk types in isolation, aligned to specific second-line functions. Given that a single process can have multiple risk types, Process Inventory serves as a coordination point across the second-line functions. Additionally, it facilitates accountability for the design and performance of mitigating controls through the identification of process owners.

To illustrate this concept, let's focus on the integrated risk assessment approach, which is used to coordinate the assessment of business continuity, compliance, and operational risks. This approach involves conducting research before engaging the first line to define the risk landscapes and requirements associated with business continuity and compliance. By doing so, a comprehensive understanding of the risks impacting processes can be achieved.

Recertification of Process Inventory

Since the Process Inventory serves as the foundation for risk assessment across various risk types, it is imperative that the inventory accurately and comprehensively represents the processes within the business unit or function. The first-line PMO resource should collaborate with the Process COE team to conduct a review of the inventory with process owners, incorporating any necessary changes and collecting Process Owner attestation. This proactive approach will help ensure that risk assessments are not hindered by quality issues.

Business Continuity Risk

US PROPERTY AND CASUALTY INSURANCE PROVIDER

In February 2021, after a series of severe winter storms, more than 4.5 million homes and businesses in Texas were without power for several days. These storms brought the coldest temperatures to North Texas in over 72 years. Homes in Texas, lacking proper insulation for such frigid conditions, relied on inefficient electric heaters, leading to a significant surge in electricity demand. Unfortunately, the Texas power grid, which was not adequately winterized, experienced a failure. Federal regulators had previously warned about the potential failure of power plants under extremely cold conditions, but the necessary actions were not taken [29].

Due to the power grid failure, there was a substantial increase in insurance claims for property and other damages, with over 456,000 claims reported and an expected payout of \$8.2 billion [30].

Later that summer, in my role as a consultant, I became involved in a project for this organization. This event exposed that their claims systems were exclusively hosted in Texas data centers. This situation required agents to capture claim details offline in spreadsheets and enter the information into the system once it was back online. This introduced a significant risk of missing claims or inaccurately entering claim data.

To address these issues and prevent similar disruptions in the future, the client initiated the project to create a comprehensive inventory of their processes. The goal was to identify similar critical processes and discover additional points of failure and implement measures to mitigate risks in the event of similar disruptive incidents.

Business continuity is the ability of an organization to maintain critical business processes during a disruptive event such as natural disasters, technological failures, cyberattacks, supply chain disruptions, pandemics, or other unforeseen circumstances. The goal of Business Continuity Risk Management is to identify potential disruptive events, defining the minimal level of operations, implementing risk mitigation measures, and developing business continuity plans.

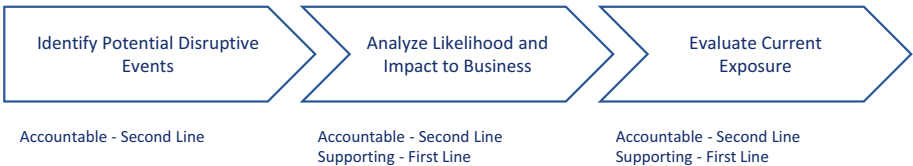


Figure 7-10. Business Continuity Risk Assessment Approach

Identify Potential Disruptive Events phase is an accountability of the second line of defense as these events may impact business units or functions across the enterprise. The second-line team should gather internal data, such as historical failures, and leverage external sources, such as industry or regulatory reports, to be as comprehensive as possible to the potential event scenarios and to estimate the likelihood of each occurrence. In this phase, a draft Business Impact Analysis document will be created, capturing details related to the impact on business operations, minimum level of operations, mission-critical operations, and disaster recovery procedures.

Analyze Likelihood and Impact to Business phase focuses on refining the assessment of the likelihood and impact of potential disruptive events on the business. The impacts should be mapped to specific processes in the Process Inventory, providing the assessment team with access to the resources those processes leverage in execution and the procedures that staff follows. The second-line team is accountable but collaborates with the first-line team, which helps identify the processes that may be affected.

Evaluate Current Exposure phase involves reviewing these potential events with existing controls in relation to the organization's risk tolerance to identify any weaknesses or gaps. The goal of this phase is to strengthen the resilience of processes, identify recovery mechanisms, and determine contingency plans. The findings are captured and maintained in the Business Impact Analysis document. The second line is accountable for this phase, but process owners play a key role in designing and implementing controls identified during this exercise.

Compliance Risk

A GLOBAL FINANCIAL INSTITUTION

This institution has faced several regulatory findings related to weaknesses in their operational and compliance control environment throughout their enterprise. In response, they have embarked on a multiyear program to address these concerns, with a focus on establishing a robust process framework.

Given this context, I was brought on to lead the Process Excellence team for the US Personal Bank. Since joining, I worked closely with the business unit leaders and their staff to create a comprehensive Process Inventory. Despite having a relatively small team, this was on track to complete the inventory by the end of a year, which is an impressive achievement.

During the process, we mapped data from their GRC repository to the Process Inventory in key business units. This exercise has helped identify opportunities to address risk data quality issues and improve overall data management practices. Furthermore, the introduction of Process Ownership has fostered a stronger culture of accountability within the first line of defense units.

Overall, this effort was aimed at enhancing operational and compliance controls, and the progress made thus far demonstrates that this framework can be effective in these complex risk challenges.

Compliance risk is the potential for organizations to violate laws, regulations, industry standards, or internal policies and procedures. Failure to effectively manage compliance risk can result in penalties, fines, legal actions, damaged reputation, and loss of customer trust.



Figure 7-11. Compliance Risk Assessment Approach

Identify Applicable Laws and Regulations is the accountability of the Compliance team within the second line, usually in partnership with the legal organization. There are a finite number of rules that govern the organization. The objective is to have a complete inventory to report to leadership and regulators, highlighting any potential gaps. This process must be led by the second line as these rules cut across organizational boundaries, especially in heavily regulated industries.

Assess Compliance Requirements and Risks begins with interpreting the regulatory rules and translating them into specific requirements or duties that business units or functions must fulfill. The second-line teams will identify and document the risks associated with noncompliance of these obligations through the risk matrix. This step is the accountability of the second-line team with support from the first-line teams.

Map Regulatory Obligations to Process involves taking the identified regulatory obligations and mapping them to specific processes within the Process Inventory. This alignment ensures Process Owners are accountable for the design and implementation of controls. This work is a collaborative effort between the first- and second-line teams.

Evaluate Existing Controls and Prioritize Mitigations focuses on reviewing the existing controls, relative to the identified regulatory obligations, for those specific processes to determine their adequacy in meeting those obligations. If any gaps or weaknesses are identified, they are prioritized for mitigation.

Operational Risk

Operational risk is the risk of doing business; it's the risk of loss resulting from inadequate or failed internal processes, people, systems, or external events. It is the potential for negative consequences arising from the day-to-day operations and activities of an organization. Operational risk includes a wide range of risks such as human error, system failures, fraud, supply chain disruptions, and other operational issues that can impact the organization's reputation, financial stability, and ability to achieve its objectives.

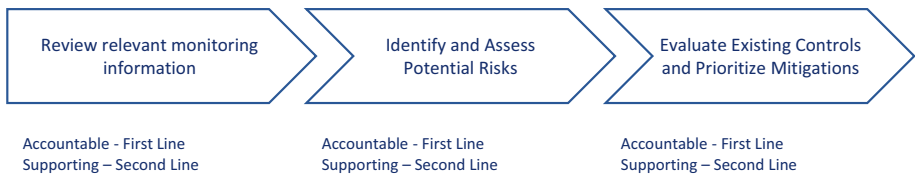


Figure 7-12. Operational Risk Assessment Approach

Review relevant monitoring information is focused on reviewing monitoring data, process performance data, issues, complaints, etc., to gain deeper insights into the performance of controls and identify emerging risks. If machine learning has been deployed to identify patterns, this would be a great opportunity to review its observations. The first-line team and the process owner must take the lead on this since they have the greatest expertise in the processes. The second-line teams should collaborate as effective challenge partners to mitigate biases that may arise from the owners being too close to the details.

Identify and Assess Potential Risks is typically conducted through collaborative workshops and focuses on a business process or group of processes identified in the Process Inventory to identify and qualify risks. The workshop participants usually include the process owners, second-line representatives for all subtypes of operational risk, technology representatives, product owners, and anyone else with expertise in these processes or who can act as an outside challenger. Risks are scored in the risk matrix, and any other pertinent information such as potential losses is captured. The first line is accountable for running this process and documenting the output.

Evaluate Existing Controls and Prioritize Mitigations is the responsibility of the first line with second-line review. In this phase, the risks identified in the previous workshops are reviewed against the existing controls in the GRC risk data repository to identify gaps or weaknesses in the current controls. Updates to existing controls or the implementation of new controls are prioritized.

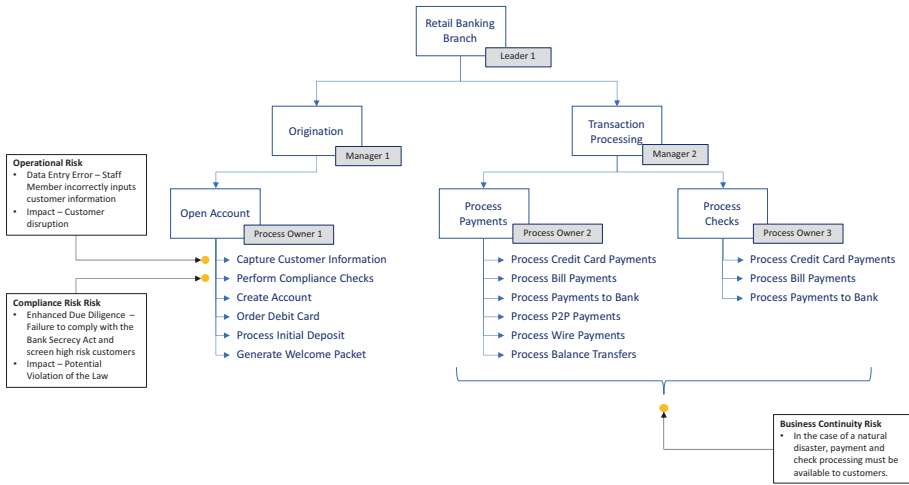


Figure 7-13. Process Inventory Risk Alignment Example

Outcome of the Integrated Risk Assessment Approach

The common theme across the risk assessment approaches for these risk types is the focus on business processes and clear accountability of the first line, the process owner, and the second line. As shown in Figure 8-5, each risk is aligned with the processes or process groups identified in the Process Inventory. At the end of these assessments, the process owner should formally attest, with the second line concurring, that these risks are valid and that they accept accountability for the implementation and performance of the controls. Once attestation has been captured, these records are entered into the GRC risk data repository.

This structured approach ensures that each process is evaluated through these assessments which means nothing will fall through the cracks. It also enables risk ratings to be aggregated to report on the risk of processes and process areas, which will focus business leaders on processes that fall outside of risk appetite to prioritize investments more effectively for mitigating actions.

This addresses the previously mentioned challenges of risk data quality issues and role clarity in the risk operating model.

Key Takeaways

- **The importance of risk management** is driven by various factors, including increasing complexity, regulatory requirements, and the evolving role of risk oversight on boards.
- **Enterprise Risk Management** is a standard methodology designed to identify potential events that may affect the entity and manage risk to be within its risk appetite to provide reasonable assurance regarding the achievement of entity objectives.
- **Opportunities to enhance risk management** include the adoption of quantitative measurements, minimizing human bias in risk assessments, improving cross-organizational coordination through end-to-end process awareness, enhancing risk data quality, and clarifying the risk accountability model.
- **Process Inventory can enable strong risk management** by aligning all risk-related activities with organizational processes. This approach has the potential to enhance the quality of risk data, strengthen the risk operating model, and leverage emerging technologies to improve risk intelligence.

In Part 3 of this book, we'll explore how to implement the Process Inventory framework. This exploration begins with the next chapter, which focuses on roles and accountabilities within the Process Inventory Operation Model.

PART
III

Implementing the Process Inventory Framework

The Process Inventory Accountability Model

In this chapter, we will start by emphasizing the importance of establishing clear roles and accountability throughout the organization when implementing the Process Inventory framework. This will provide you with a comprehensive understanding of the pivotal roles played across the organization in developing comprehensive and accurate models, which will ultimately result in significant organizational benefits.

We will also explore the significance of creating a centralized Process Center of Excellence (COE) team, offering valuable insights into the specific processes they undertake to develop and maintain the models and data necessary for this strategic approach.

Furthermore, we will delve into key collaboration points across organizational units, including business, technology, customer experience, and risk management. This exploration will help you grasp the integral roles these domains play in driving the success of this strategy.

Driving Employee Engagement at Spotify: Balancing Autonomy and Accountability

Spotify, with a revenue of €11.7 billion and a workforce of 9,473, is a prime example of a company that strategically leverages its organizational structure to enhance employee engagement and drive business performance [1]. Spotify's business model and strategies require rapid growth to optimize operational effectiveness and maximize competitive advantages. This growth involves innovation, such as new product development for enhancing customers' mobile listening experiences [2].

Their growth is in part powered by how they balance autonomy and accountability within their organizational culture, creating an environment where innovation thrives and employees are motivated to contribute their best efforts. This approach has led the company to secure the top spot in *Newsweek's* "America's Most Loved Workplaces" in 2021 [3].

The heart of Spotify's culture lies in its "controlled chaos" philosophy, embodied by its band manifesto [4]. Like a band, they're dependent on each other to create the best audio experience, they need to be in sync, and like any successful band, they have a set of rules, a band manifesto, that keeps them focused on where they want to go and guides how they get there.

This analogy emphasizes the interdependence and synchronization among employees, fostering a sense of shared purpose. The core values of innovation, collaboration, sincerity, passion, and playfulness further define Spotify's culture, setting the stage for an environment where employees are encouraged to explore and innovate.

Spotify's model underscores the need for alignment to effectively enable autonomy while balancing controls [5]. The company had to make explicit choices in its operating model, ways of working, and culture that address the three core tensions between individual autonomy and organizational goals.

Leadership's role is in reinforcing alignment. Prior to launching projects, leadership emphasizes establishing alignment on objectives and goals. This shared understanding provides the anchor for autonomous decision-making, exemplifying how strong alignment among teams can drive collaboration.

Central to Spotify's approach is the organization of employees into autonomous, self-organizing units known as "squads." These squads, comprising cross-functional teams, are accountable for a discrete aspect of

the product and own it from cradle to grave. These squads, which are no more than eight people, possess the authority to decide what to build, how to build it, and with whom to work with to make the product interoperable. They are organized into a light matrix called a tribe. Tribes comprise several squads linked together through a chapter, which is a horizontal grouping that helps support specific competencies such as quality assistance, agile coaching, and web development. The chapter's primary role is to facilitate learning and competency development throughout the squads.

This autonomy empowers employees to take ownership of their work, experiment with innovative ideas, and foster a heightened sense of accountability as they directly impact project outcomes. The company values the speed of iteration over the pursuit of perfection, fostering a culture where experimentation is encouraged, and failure is embraced as a learning opportunity. To enable this, they supported this strategy with measurable objectives, consistent measurement of progress toward those goals, feedback systems to monitor activities along the way, and appropriate consequences for reaching or failing to reach the goals. Results are visible both through internal reviews and customer feedback, and squads are expected to fully understand successes and failures. Squads go through postmortem analyses of failures to ensure learning. Every few weeks, squads conduct retrospectives to evaluate what is going well and what needs to improve.

The Spotify model, which integrates autonomy with accountability, serves as a blueprint for driving employee engagement through a culture of innovation. By encouraging ownership, enabling learning, and fostering alignment, Spotify creates an environment where employees are motivated to contribute creatively and collaboratively. This dynamic model serves as an inspiration to organizations seeking to maximize engagement while maintaining focus on strategic goals.

Establishing a Culture of Accountability

The culture of an organization largely determines business performance. As with the Spotify example, this includes many factors but especially alignment to organizational goals, accountability, and culture of innovation. According to a recent Gallup poll [6], accountability is an area that needs some improvement as they report that only 14% of employees feel their performance is managed in a way that motivates them, only 21% feel their performance metrics are within their control, and that 70% of employees feel their managers aren't objective in how they evaluate their performance.

Defining a clear accountability framework is particularly crucial for the success of implementing the Process Inventory framework to drive digital transformation or other high-priority business goals. This chapter focuses on the human side of implementing this framework. As you'll see, implementation will require coordination across a wide set of stakeholders and organizational units.

When defining the Process Inventory strategy, in addition to detailing the short- and long-term ambitions of the framework and its use cases, it's critical to include an **operating model design** that outlines the necessary processes, roles, accountabilities, governance, and culture required to facilitate its implementation.

This operating model needs to explicitly **define goals and metrics** that cascade throughout the organization, ensuring that individual and team objectives align with the organization's goals and are harmonized with the aspirations of implementing this framework.

A substantial aspect of achieving a clear accountability framework is being meticulous in **setting expectations**. Employees need to have a clear understanding of what's expected of them and how their performance will be measured. Managers will need to gain commitment from team members regarding these expectations, providing them with the opportunity to ask clarifying questions and ultimately take ownership of these accountabilities.

Leadership plays a pivotal role, as any cultural changes must be reinforced from the top. They must articulate in their communications the compelling change story that motivates the organization. They also need to demonstrate in their actions that this is a strategic endeavor and that they, along with the rest of the management team, are committed to the transition and will support employees along the way.

A critical aspect of this blueprint is a **comprehensive assessment** of the prevailing organizational culture and human resources. This evaluation enables strategic decisions related to hiring, outsourcing, training, and other initiatives to be identified and effectively planned.

Process Inventory Framework Operating Model

As mentioned earlier, a key to success in this initiative is to detail the operating model, define processes and roles, and set clear expectations for this framework to reach its potential. To highlight what that looks like in practice, the remainder of this chapter will be dedicated to going through the roles in Figure 8-1 for delivering and maintaining the models and data in this framework.



Figure 8-1. Process Inventory Partial Operating Model

It's important to note a couple of points relative to this model:

- **Illustrative Model:** Each organization has a unique organizational structure, which the following model may not be aligned with. It's important to leverage these illustrative insights and define the model in the context of your organization.
- **Delivering and Maintaining Models and Data Only:** The use cases covered in Chapters 4–7 and the Transformation Management Office covered in Chapter 3 are excluded from this discussion as many of the roles and accountabilities were covered in those chapters. This chapter will be focused on the core roles in creating and maintaining quality models and data.
- **Not Exhaustive:** This model can extend significantly to other parts of the organization, especially as it relates to the management and integration of metadata from other sources. The purpose here is to give a representative sample to extend from.

This model has been deployed in numerous large organizations, through which this approach has been continually refined to increase effectiveness. The processes, roles, and accountabilities have been proven to scale and sustain the Process Inventory framework, positioning it to deliver value to the organization.

Process Center of Excellence (COE)

At the core of an effective Process Inventory strategy is the Process COE. This organization is responsible for defining the strategy, facilitating model creation through engagement with stakeholders, managing the platform and data infrastructure, sustaining the models through governance to remain reflective of the organization, and more. In this section, we'll delve into the details of the role this group performs.

Organizational Structure

Figure 8-2 is an illustrative organizational structure that is meant to showcase the roles and reporting structures, rather than the headcount, as that will depend on factors such as the ambition of the strategy and the size of the organization.

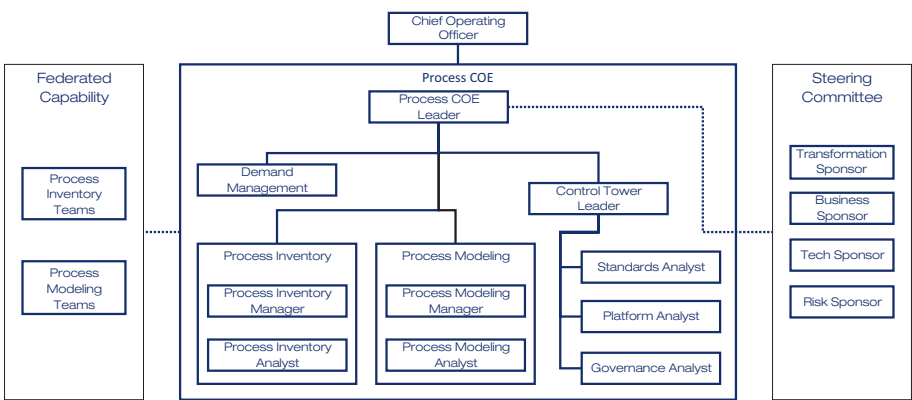


Figure 8-2. Process COE Organizational Chart

A couple of items to highlight here include

- **Placement Within the Organization:** In this illustration, the group reports to the chief operational officer, which makes logical sense as the goal of this effort is to streamline operations by adopting digital technologies. I have also seen this group roll up at other organizations to technology, risk, and transformation offices. I believe any of these options will work if this team isn't hindered in their purview or focus and is able to accomplish their vision.

- **Steering Committee:** Building and maintaining a diverse set of senior stakeholders is a best practice as they can provide you with their insights on the organization's highest priorities, ensuring that the team is focused on what's most important.
- **Federation:** In some large organizations, business units may want to create their own Process Inventory and process models. This is perfectly acceptable as long as they comply with all modeling standards and quality review practices, ensuring that the models across the organization remain consistent. A close and collaborative relationship across these federated teams and the Process COE should be maintained.

Process COE Process Inventory

Figure 8-3 represents the Process Inventory for the organization. I'll leverage this to cover the details of the accountabilities, processes, and skills for each part of the organization.

Process COE				Manage Process Control Tower	
Lead Process Delivery	Manage Process Demand	Deliver Process Inventory	Deliver Process Models	Maintain Process Standards	Govern Process Assets
Maintains Process Inventory Strategy	Evaluate New Modeling Requests	Construct Process Inventory	Construct Process Models	Maintain Modeling Standards and Checklists	Perform Quality Reviews
Conduct Stakeholder Outreach	Maintain Estimation Model	Conduct Periodic Process Inventory Refresh	Conduct Periodic Process Models Refresh	Delivery Process Training	Manage Attestation Library
Advise on Process Inventory Use Case Implementation	Maintain Demand Forecast Model	Perform Change Requests	Perform Change Requests	Maintain Process Methods	Publish Models
Leads Process CoE Delivery	Allocate Resources	Fulfill Ad hoc Requests	Fulfill Ad hoc Requests	Manage Platform and Data	Retire Models
Measure Impact Across the Organization	Create Reports for Stakeholders			Administer Modeling and Data Platform	Initiate Model Refresh Reviews
				Perform Data Migrations	Maintain Governance Workflows
				Perform Data Quality Checks	
				Maintain Data Library Architecture	
				Design Reports	

Figure 8-3. Process COE Process Inventory

Lead Process Delivery

Accountabilities

The leader of this group is accountable for the success of the effort. They must build and maintain relationships with senior stakeholders to define the Process Inventory strategy and ensure alignment with the organization's goals. This leader must also consult with other stakeholders on leveraging the team's assets to drive value through the implemented use cases. Additionally, the person is responsible for guiding the team to deliver quantifiable benefits, through a commitment to this framework, for the organization. They are accountable for executing the processes in Table 8-1.

Processes

Table 8-1. Lead Process Delivery Processes

Process Name	Process Description
Maintains Process Inventory Strategy	Works with a broad set of stakeholders to document the organization's goals and then define the Process Inventory strategy, which includes scope, processes, tools, and model types. The strategy should be continually revisited as the organization's strategy and priorities change.
Conduct Stakeholder Outreach	Maintains an ongoing relationship with key stakeholders to identify their needs and critical priorities. Works with them on solutions for how this framework can address their challenges.
Advise on Process Inventory Use Case Implementation	Advises stakeholders to define use case methodologies for leveraging process modeling assets to achieve their goals. This leader actively champions the use of this framework to ensure maximum value is derived from it.
Leads Process COE Delivery	Sets direction for the team, guides them to deliver high quality, hires/trains their team, reviews critical deliverables, and addresses critical issues.
Measure Impact Across the Organization	Defines metrics to measure impact and maintains a scorecard to share with key stakeholders to demonstrate the business value this framework is having.

Required Competencies

The ideal candidate for this role should exhibit a genuine passion for the subject matter, driven by a desire to create a meaningful impact throughout the organization. They should possess extensive experience in successfully implementing process management frameworks, methodologies, and tools. With strong leadership skills, they should have the ability to guide and motivate their team while also exerting influence across various levels of the organization.

Their strategic mindset should enable them to connect the dots and proactively identify opportunities to deliver value. Furthermore, a solid understanding of the business landscape coupled with broad experience across diverse functional areas will contribute to their effectiveness in this crucial role.

Manage Process Demand

Accountabilities

This team is accountable for managing a consistent level of demand for the modeling teams. This includes evaluating, prioritizing, and sizing incoming modeling requests to ensure alignment with organizational goals and resource availability. They work closely with managers within the Process COE to confirm resource availability and schedule the work appropriately to maintain a steady flow of effort. They continuously monitor and adjust demand to optimize resource allocation. Ultimately, their role contributes to keeping key stakeholders satisfied that their priorities are being effectively addressed. They are accountable for executing the processes in Table 8-2.

Processes

Table 8-2. Manage Process Demand Processes

Process Name	Process Description
Evaluate New Modeling Requests	Establish a standard process and intake form for assessing new work requests. This team must deep dive into the proposed scope, align it with the framework to quantify resource needs, and perform prioritization.
Maintain Estimation Model	Construct and maintain an estimation model that assists in accurately determining the necessary duration and resource effort, anchored in estimation criteria such as the organization's scale and the quantity of process areas. After work efforts are completed, document the actual effort expended, and continuously analyze and refine the model to further enhance accuracy.
Maintain Demand Forecast Model	Implement a model that accurately forecasts demand and supply for every week in the future. This approach will greatly improve visibility into capacity constraints and resource availability.
Allocate Resources	Collaborate with managers within the Process COE to confirm resource assignments for upcoming projects. Allocate resources to these efforts and schedule kickoff calls with key stakeholders.
Create Reports for Stakeholders	Maintain an aggregated status of all modeling efforts, including overall color status, progress toward committed milestone dates, percent completion for deliverables, and identification of critical issues. Distribute these reports to key stakeholders.

Required Competencies

The demand management team plays a critical role and requires strong project management skills for assessing and prioritizing initiatives, as well as efficiently allocating resources to meet stated goals. Proficiency in data analysis and reporting is crucial to maintain an accurate effort estimation model and demand forecast. Effective communication and collaboration abilities are essential as they act as a bridge between initiative stakeholders and the modeling teams. Additionally, strong stakeholder management skills are vital for crafting status reports that effectively address the needs and expectations of various stakeholders.

Deliver Process Inventory and Deliver Process Models

These teams are focused on building and maintaining their respective models. I'm combining them into a single section as their work is similar while focused on different model types.

Accountabilities

These teams are accountable for the delivery of accurate and comprehensive Process Inventory documentation and process models, respectively, for the scope that they are assigned. Adhering to established quality practices and complying with modeling standards are imperative aspects of their role. Their accountability extends to collaborating with stakeholders across various organizational levels, ensuring that the final work products are satisfactory to all and that required attestations are collected. They are accountable for facilitating effective sessions to make efficient use of stakeholders' time. These teams are accountable for executing the processes in Table 8-3.

Processes

Table 8-3. Deliver Process Inventory and Deliver Process Models Processes

Process Name	Process Description
Construct Process Inventory	This involves collaborating with an organizational unit for the first time to inventory their processes, identify process owners, align metadata, and collect attestations.
Construct Process Models	This is focused on working with process owners to model how the process is executed, with all required metadata aligned. These models must comply with quality standards and receive attestations.
Conduct Periodic Process Inventory Refresh	This involves engaging with the organizational unit periodically, as the timing defined by policy, to review the Process Inventory and adjust based on any process or process owner changes.
Conduct Periodic Process Models Refresh	The focus is on engaging with process owners periodically, as the timing defined by policy, to review and update process models to reflect any changes in the environment.
Perform Change Requests	The business or other stakeholders may request changes to the Process Inventory or process models based on new initiatives, organizational changes, or other drivers. This team will engage with stakeholders to make the necessary adjustments and collect attestations.
Fulfill Ad Hoc Requests	As these assets become an anchor point for use cases throughout the organization, stakeholders may need clarification or data from the completed models. This is focused on supporting those requests.

Required Competencies

Modelers must possess a strong command of process modeling techniques and industry standards, demonstrating proficiency in utilizing process modeling tools. Effective communication skills are essential, enabling them to lead sessions with stakeholders, pose clarifying questions, and ensure the accuracy of models while optimizing the use of time. A comprehensive grasp of cross-functional domains, encompassing business, technology, risk, and other areas, empowers them to fully comprehend and critically question the subject being modeled. Their attention to detail and commitment to delivering high-quality work products are crucial for maintaining the integrity of the models. Their appreciation for governance principles and adherence to standard methodologies further enhance their ability to contribute effectively to the modeling process.

Control Tower

Accountabilities

The control tower team is accountable for defining and maintaining documented standards and ensuring that the assets published in the Integrated Operational Repository adhere to strict quality standards. They achieve this by conducting training for modeling teams, maintaining and distributing quality checklists, and serving as the last line of defense for quality by conducting reviews on models and data assets. This team is also responsible for architecting, administering, and configuring features in the Business Process Analysis platform, as well as managing the migration and quality of data in the platform. Additionally, they facilitate governance processes to ensure that accuracy and quality are sustained over time. They are accountable for executing the processes in Table 8-4.

Processes

Table 8-4. Control Tower Processes

Process Name	Process Description
Maintain Process Standards	
Maintain Modeling Standards and Checklists	The focus is on building and maintaining the standards for Process Inventory, process models, and the associated metadata, which are based on industry norms and the needs of the organization. These standards are translated into quality checklists so that modeling resources can conduct quality reviews on their own and others' models.
Delivery Process Training	Training is essential for new modelers in the Process COE and federated modeling teams. This process is dedicated to delivering comprehensive training to ensure that these resources possess a deep understanding of how to effectively create models.
Maintain Process Methods	Methods outline how modeling should be conducted, encompassing activities, templates, quality checkpoints, and more. This team continuously learns from completed endeavors to enhance the methods, making them more effective over time.
Manage Platform and Data	
Administer Modeling and Data Platform	This group is responsible for all administrative tasks, such as user entitlements, configurations, performance monitoring, license management, and more. This is a higher-level process that further breaks down into many subprocesses.
Perform Data Migrations	Data from authoritative data sources needs to be migrated and periodically refreshed on the platform. This process handles the refreshes and addresses any data conflicts that arise.

(continued)

Table 8-4. (continued)

Process Name	Process Description
Perform Data Quality Checks	Periodically, reports must be executed to highlight any data inconsistencies or conflicts in the models and data libraries. These processes not only identify issues but also work to resolve them.
Maintain Data Library Architecture	Migrated data from authoritative sources needs to be stored in libraries for modelers to perform associations within their models. This process designs these libraries and updates them as required.
Design Reports	This team collaborates with stakeholders to capture their reporting requirements and then configures custom-built reports in the Business Process Analysis platforms.
Govern Process Assets	
Perform Quality Reviews	As part of the processes to construct Process Inventory and process models, this quality team will conduct a final quality review of the models.
Manage Attestation Library	A library for storing attestations needs to be maintained so that each artifact can be traced back to the owner who approved it. This process is focused on managing this library for completeness and accessibility.
Publish Models	Once models have completed the necessary quality checks and obtained all required attestations, they can be published to the production environment within the Business Process Analysis platform.
Retire Models	There are several reasons models may need to be retired, such as changes to the process, elimination, business reorganization, or outdated content. This process is focused on removing these models from the production environment and archiving them.
Initiate Model Refresh Reviews	To maintain the accuracy of models, they need to be periodically re-attested. This group is responsible for maintaining accurate date information on models that require re-evaluation. They will collaborate with the modeling teams to conduct stakeholder reviews, update model content, and obtain necessary approvals.
Maintain Governance Workflows	Many of the governance processes require facilitation through workflows, either within the Business Process Analysis platform or using external tools. These workflows play a critical role in coordinating and monitoring governance activities. The primary focus of this process is to establish and maintain these workflows, implementing ongoing enhancement as governance processes are refined.

Required Competencies

This team must possess meticulous attention to detail and a commitment to quality. They need the courage to reject poor quality, even if it means challenging individuals with higher seniority. They must be experts in modeling standards and possess critical thinking skills to define standards that are tailored to the organization. Additionally, they should be capable of designing and executing rigorous governance processes to ensure nothing is overlooked.

The team must also have the technical acumen to architect and configure the structure of models and data within the Business Process Analysis platform. They require data management competencies for performing data migrations and executing data quality processes. Moreover, they need to be dedicated to customer service, innovation, and continuous learning from past efforts to enhance methods and standards going forward.

Furthermore, this team requires strong communication and stakeholder management skills to effectively design and deploy training as well as other communications about the platform.

Business/Functional Areas

Business units, which are profit-generating segments of the organization, and functional areas that support the business both play a critical role in the success of these frameworks as they are ultimately accountable for the content of the models. To be successful, these accountabilities need to be detailed clearly and broadly communicated, especially in large organizations where many individuals will perform these duties. The most important roles are the following.

Process Owner

Process owners hold the accountability of providing precise process information to modeling teams, ensuring an accurate representation of the inventory of processes and their execution details. Their responsibility extends to formally attesting to the accuracy of completed models.

As the Process Inventory is derived from the organizational structure, a distinct hierarchy of ownership can be traced to senior leadership. Process owners and their teams, actively involved in the day-to-day execution of these processes, inherently possess deep knowledge, eliminating the need for extensive training to be effective. Their responsibility lies in upholding the clarity and precision of the models and promptly notifying the Process COE of significant changes, ensuring timely updates to the models.

Risk Owner

Assuming that risk management is a prioritized use case within the organization, as defined in the Process Inventory strategy, the role of the risk owner becomes significant in identifying risks in processes and defining appropriate controls. A risk owner may be the same individual as the process owner, or in some organizational units, a central risk management team may fulfill this role. Regardless, risk owners need to identify risks and ensure that they are appropriately aligned in the Process Inventory taxonomy and accurately represented in process models.

Product Manager

Product managers hold the responsibility for steering the strategy and success of a product throughout its life cycle. Through market research and understanding of customer expectations, they establish business goals and are accountable for delivering results for each product they own. They manage a product roadmap, which details new features and functionalities to drive the product's objectives.

Since products are supported with numerous processes across the organization, the features and functionalities directly correlate with how processes are designed. Product managers and their teams frequently play a key role in defining the behavior of future state processes that are being implemented through change initiatives. It's important that they understand the landscape of processes which support their products so they can provide direction for improvement or optimization.

Subject Matter Experts

Subject matter experts play a critical role within the Process Inventory framework, bringing deep knowledge and insights that contribute to accurate process models. Beyond that, they can provide valuable coaching to modeling teams navigating organizational units, offering effective challenges to what process owners are describing. They typically possess a perspective on how to enhance the process environment and can identify opportunities for improvement or optimization.

Data Stewards

In certain organizations, the role of data stewards is employed within specific business or functional units. This role holds the responsibility of identifying and documenting crucial data elements, including definitions, rules, governance processes, linkage to data models, ownership, custodianship, and more,

utilized by those organizational units [7]. Serving as a bridge between the organization's data users and technical teams, data stewards facilitate the alignment of business needs with technical requirements.

If your organization features this role and data management is a prioritized use case for leveraging the Process Inventory framework, data stewards assume a pivotal role in modeling the process environment. When processes are modeled, they contribute to identifying data attributes leveraged during process execution. These attributes are captured in the data objects of the process models, and within the Business Process Analysis platform, they can be mapped to the enterprise glossary or data model. This practice delivers a clearer depiction of how data is utilized across processes, fostering enhancements in data governance, data controls, data quality, data architecture, ownership, accountability, and more.

Technology

The intersection between processes and technology, combined with the adoption of a digital agenda, necessitates a strong partnership between the Process COE and various segments of the technology organization, where alignment on goals is key. IT Asset Management and Enterprise Architecture represent a subset of those partnerships.

IT Asset Management

The IT Asset Management team is accountable for tracking, monitoring, and optimizing the entire life cycle of IT assets within the organization, encompassing hardware, software, licenses, and other technology-related resources. The Process Inventory initiative relies heavily on the accuracy of this data as it must be migrated to the Business Process Analysis platform to facilitate mapping between IT assets and activities within processes. While this holds true for any metadata integrated into process libraries, given the paramount role of this relationship in digital transformation, the quality of this information becomes critically significant. This necessity underscores the need for a close collaboration across groups.

Enterprise Architecture

Enterprise Architecture is the discipline by which organizations standardize and organize their IT environment to align with business goals. Chapter 6 covered several ways in which the Process Inventory framework can enable Enterprise Architecture designs. When designing the accountability model for the Process Inventory framework, leaders from the Process COE and the

Enterprise Architecture organization should closely work together to define areas of close collaboration which will be dictated by their respective strategies.

Many Enterprise Architecture organizations are responsible for modeling the technology environment. An Enterprise Architecture platform, which has similar capabilities as the Business Process Analysis platform, can be used to document the architecture framework, define IT blueprints, document solution architectures, and more. The models across both environments need to harmonize so that stakeholders can clearly understand how the technology architecture supports business processes. Processes within each organization must include collaboration points to ensure integrity.

In addition, if Enterprise Architecture adopts system sequence diagrams at scale with a goal of sustaining these assets, then an operating model very similar to the Process COE outlined earlier in this chapter would be needed to build and maintain the connection between process activity and interactions across downstream systems. Maintaining integrity across both environments can deliver powerful benefits for streamlining and making the IT environment cost-efficient.

Customer Experience (CX)

In the realm of digital adoption, enhancing customer experience and engagement emerges as a significant driver. This concept gains validation from a survey that shows 70% of senior executives consider CX a top priority [8]. Consequently, numerous organizations are creating dedicated customer experience teams to advance the customer agenda. These teams focus on customer research, identifying opportunities, and crafting solutions to elevate the overall customer experience.

As described in Chapter 2, these teams will create models, specifically customer personas and customer journeys, to replicate and study the current and desired experience of customers. To enhance root cause analysis for opportunities, it's crucial to link interaction points in a customer journey to specific processes within the Process Inventory that are executed during those interactions.

Close collaboration between the CX and the Process COE is needed to identify those linkages. The CX team should consider building and maintaining their models in the Business Process Analysis platform to gain many of the benefits the platform brings such as dynamic impact analysis, reporting, and transparency to processes directly supporting customers. If so, then collaboration with the Process COE control team is needed to harmonize how model governance is implemented.

Risk

Second Line of Defense

I extensively covered the significant opportunities for Process Inventory to contribute to a robust risk management environment in Chapter 7. If prioritized within the organization's strategy, this goal necessitates the definition of multiple collaboration points across the organization.

In this section, I would like to highlight a couple of areas within the second line of defense, which holds the responsibility for overseeing and monitoring risks. The second line encompasses various suborganizations, typically aligned with the risk taxonomy such as operational risk, compliance risk, cybersecurity risk, etc.

If the Process Inventory taxonomy is integrated into the governance, risk, and compliance (GRC) tool—a platform for risk data and process management—a tight collaboration between the governance teams within process and the second-line function that manages the platform becomes paramount. This collaboration ensures the integrity and quality of data sets are upheld. Additionally, the second-line risk functions should conduct reviews of risks and controls aligned with processes to validate their accuracy and alignment with policy.

Key Takeaways

- **Establishing clear accountabilities** is critical to drive business value leveraging this framework, which will require collaboration across many diverse organizational units. Designing explicit processes, roles, accountabilities, governance, and a unified culture across organizational boundaries is essential for success.
- **Process COE** is a central body that articulates strategy, defines standards, builds process models, and governs assets. Establishing this COE with clear roles and processes is crucial for delivering comprehensive and accurate models that contribute to business value over time.
- **Defining partnerships and collaborations** across diverse domains, such as technology, customer experience, and risk, and aligning on objectives are crucial for crafting a comprehensive end-to-end model for success.

In the next chapter, we will delve into the methods and modeling rules required to create high-quality models that deliver business value.

Process Methods and Modeling Rules

In this chapter, you will gain a deep understanding of why modeling the processes of an organization through Process Inventory and process models is of paramount importance. We will delve into the guiding principles that maximize the value an organization can derive from these models.

We will introduce you to an industrialized approach for creating models, one that not only produces high-quality output but also optimizes the use of valuable resources, such as process owners and other stakeholders. This will include best practices for conducting modeling workshops, whether for current state or future state processes.

You will learn best practices and industry standards for creating clear and consistent models, which include the significance of maintaining the integrity of the relationship between Process Inventory and process models.

Finally, we will cover the role of peer reviews and final quality checks in the delivering model that are accurate and comply with modeling standards.

Unilever's Digital Transformation of Supply Chain Through Modeling Processes

Unilever plc, a British multinational consumer packaged goods company founded in 1929, has established itself as a global leader with a portfolio of renowned brands including Axe, Ben & Jerry's, Dove, Knorr, and more. Operating in over 190 countries, Unilever delivers high-quality products to more than 2.5 billion consumers daily, underscoring the significance of an integrated and connected supply chain in managing an impressive array of 77,000 stock keeping units (SKUs) across the globe [1].

As challenges within the consumer goods sector mount—ranging from inflation-driven pricing surges to material and labor shortages—Unilever embarked on a digital transformation journey to maintain a competitive edge [2]. With a market value of \$55 billion, the company recognized the power of data insights in predicting the future and responding to evolving consumer demands [3].

One of Unilever's key objectives was to harness the capabilities of digital technology to revolutionize its supply chain. This task is further complicated by the fact that they oversee more than 220 owned factories, manage over 3,300 discrete product lines, and maintain more than 900 partnerships [1].

Customers now expect customization, on-demand products, and brands with a clear purpose. This vision compelled Unilever to commit to digital transformation, underscoring the significance of becoming data-driven to meet evolving customer expectations.

At the heart of Unilever's transformation lies the concept of the “digital twin.” A digital twin is a next-generation digital model of a physical environment. In Unilever's case, it's a digital replica of their factories. This digital twin, powered by the Internet of Things (IoT) and intelligent edge services within the Azure IoT platform, captures data from sensor-equipped machines and processes. The digital twin models the physical plant, providing real-time insights into operations [3].

By creating this digital twin, Unilever gains unprecedented visibility into its supply chain. The digital model simulates all aspects of the plant, from individual machines to entire processes. Advanced analytics and machine learning algorithms are applied to the collected data to predict outcomes based on historical patterns. This empowers Unilever to readily anticipate and respond to various challenges, ensuring optimal efficiency, resiliency, and flexibility. The strategy aligns with the growing trend of utilizing data streams from the Internet of Things to drive operational enhancements across various industries [4].

Through the modeling of processes and the harnessing of advanced technologies, Unilever has not only optimized its supply chain operations but also gained the ability to predict and respond to market dynamics. This approach has facilitated cost reduction, improved efficiency, and enhanced decision-making, positioning Unilever as an industry trailblazer at the forefront of digital innovation.

The impact of this digital transformation has already become tangible. Unilever's digital twin empowered operators to exercise meticulous control over soap-making machine moisture levels, resulting in enhanced consistency. Furthermore, by analyzing data on production times for liquid batches, such as shampoo and detergent, the digital twin optimizes the order of processes to maximize production capacity [3].

The Importance of Models

Organizations are complex systems; in fact, they are systems made up of complex systems. A complex system is defined as being made of many interacting components that exhibit emergent behavior, meaning that the behavior of the system cannot be explained solely by the behavior of its individual components. As organizations grow, they don't just get bigger, but they get more complex from specialized resources and tight integration internally and with external parties. Each component is and should be working toward the corporate strategy, but in many cases, they adopt a goal of sustaining itself.

The study of complex systems is prominent in many different fields such as biology, ecology, economics, sociology, and computer science. These fields create models to predict behavior, apply control, and optimize systems. The overarching goal is to make these systems understandable to humans, and increasingly machines, so that pressing problems can be triaged and informed business decisions can be made.

As demonstrated in the Unilever case study, modeling the environment through processes is necessary to maximize the utilization of digital technologies to achieve operational excellence and competitive business value.

This chapter is focused on creating models, through the Process Inventory framework, within organizations to achieve a deeper level of transparency. As I highlighted in Chapter 2, there are multiple models that come together to fully describe an organization. Detailed in the following include the approach for creating the models, standards to adhere to a high level of quality, best practices, and guiding principles.

Guiding Principles for Success

Prior to delving into the specifics of individual creation processes, it's essential to establish a bedrock of crucial guiding principles. These principles are pivotal in enhancing the likelihood of achieving success:

- **Alignment with Strategic Goals:** The models generated by this process should seamlessly integrate with the organization's requirements, as highlighted in the digital strategy covered in Chapter 3. The documentation and aligned metadata need to support the digital aspirations, use cases, and overarching vision of the organization.
- **Prioritizing Quality:** Quality takes center stage—these models come together to form a repository of information, offering a comprehensive view of the organization. As stakeholders begin to utilize this repository within their roles, encountering significant quality discrepancies could undermine their confidence in the initiative. Overcoming negative perceptions of inadequate quality presents a considerable challenge. For each model, I will outline a construction approach. Within these methodologies, I will demonstrate multiple stages for quality reviews and stakeholder attestation. Adhering to these quality checkpoints is of utmost importance to prevent potentially costly or detrimental errors. A best practice involves conducting internal quality reviews before seeking stakeholders' validation. This approach avoids the need for repeated validations if quality issues emerge.
- **Clarity and Consistency:** Models must be documented with a level of clarity that even individuals unfamiliar with the process or its domain can grasp the depicted actions. Furthermore, these models should adhere to consistent standards, ensuring uniformity across diverse organizational units. Accomplishing this entails the establishment and upkeep of comprehensive standards and checklists that seamlessly integrate into the quality assurance processes.
- **Ownership and Formal Attestations:** Every model requires a designated owner who can vouch for the accuracy and comprehensiveness of the models. As individuals transition between roles or depart, it is crucial

to establish clear ownership within the organizational hierarchy, ensuring that a replacement owner can be designated, if necessary. The process of attestation should be conducted formally and documented for audit purposes. Capturing attestations can range from straightforward email confirmations stored on a shared drive to utilizing a workflow tool. In case concerns about quality arise, maintaining a traceable record of the attestation provider's identity becomes indispensable.

- **Balancing Excellence and Practicality:** This sentiment underscores the necessity for efficiency. The stakeholders with whom the Process COE team engages have full-time day jobs. Hence, optimizing their time is paramount. I've witnessed endeavors of this nature stagnate in prolonged analysis without any substantial progress. Therefore, it's essential to strike a balance between aiming for perfection and achieving practical results.
- **Progressive Strengthening Through Adoption:** Although the models may not achieve perfection upon their initial creation, their refinement will evolve with time as broader stakeholders embrace and integrate them into their operations.
- **Sustaining Model Relevance:** Organizations are dynamic entities, undergoing frequent changes. An inevitable question raised when presenting this concept to new stakeholders revolves around the models' ability to maintain accuracy over time. The concise response entails implementing a periodic attestation process, which at a minimum should be done annually. This process entails presenting the model to its owner(s), who identify alterations and subsequently revalidate its accuracy. However, I regard this as the baseline approach. Ideally, if these models are leveraged across a broad set of use cases, the organization's vested interest in accuracy simplifies the integration of updates.
- **Fostering Continuous Enhancement:** Maintain a culture of continuous enhancements by reviewing lessons learned to refine and elevate the methodology for constructing models. Actively solicit feedback from users and stakeholders, channeling their insights to iteratively elevate the efficiency and efficacy in documenting the process environment.

Process Inventory

The main objective of creating the Process Inventory is to identify all the processes within the specified scope of the organization. This forms the fundamental basis for the alignment of resources and operating information. In this section, I will delve deeper into the approach of engaging with the business to create this model while also emphasizing the importance of standards and best practices for success.

For well-established business domains, this task is typically carried out based on the current state. This approach is highly efficient as it involves interviewing stakeholders about the processes they currently execute. This shouldn't require additional research as stakeholders are expected to have expertise in their respective processes. However, in situations where the business area is new or undergoing a redesign, the focus can shift toward the future state. This might take more time, especially if the leaders haven't fully conceptualized their future processes. When interacting with leaders, you'll need to ensure clarity on the focus, which will help you create accurate timelines and resource estimates as needed.

Construct Process Inventory

When creating the Process Inventory, the following approach as depicted in Figure 9-1, which has been demonstrated in multiple organizations, has been proven to be efficient and to produce high-quality output due to the rigor of multiple quality checkpoints.

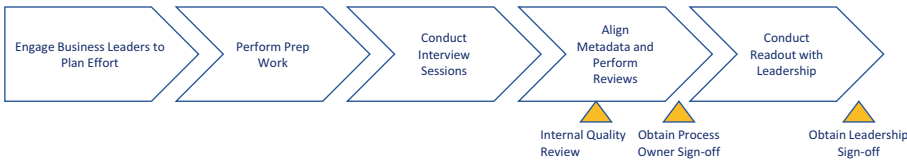


Figure 9-1. Construct Process Inventory

Engage Business Leaders to Plan Effort

This step assumes that you have already secured broad leadership buy-in, from the chief transformation officer and other equivalent senior leaders, for this initiative. The purpose of this phase is to connect with organizational unit leaders, who might not possess a comprehensive understanding of this work. The aim is to educate them about the effort and formulate a plan outlining the timeline and the precise resource requirements from their respective departments. The objective is to gain their authorization to involve their teams in this initiative. The activities in this phase are outlined in Table 9-1.

Table 9-1. Engage Business Leaders to Plan Effort Activities

Activities	Outcome(s)
• Prepare the Benefits Case: If this leader is not entirely familiar with this effort, it's crucial to educate them on the rationale and benefits behind this exercise. Whenever possible, align the case with their priorities. • Develop an Estimated Timeline and Resource Hours: Excess capacity is in short supply for business stakeholders. Creating and maintaining an estimation model is important, enabling you to calculate the effort needed and be efficient in the use of your time. • Validate Engaged Resources: While the organizational chart can largely provide this, confirming the appropriate resources for engagement is important. Identify any resource scheduling limitations that need to be navigated. • Identify Assets for Prework: In many instances, process details can be gleaned from existing documentation. To conserve valuable stakeholder time, request a list of resources that can provide a robust initial draft. This may include organizational charts, roles and responsibility documents, procedure manuals, and more.	• Permission to Engage with Resources • Agreed Upon Timeline

Best Practices

- **Strive for Precision in Estimating Timelines and Resource Commitments:** Given the scale of large enterprises, Process Inventory collection may involve numerous organizational units. Construct and maintain an estimation model that assists in precisely determining the necessary duration and resource effort, anchored in estimation criteria like the organization's scale and the quantity of process areas. After completing each area, document the actual effort expended, and continuously analyze and refine your model to further enhance accuracy.
- **Tailor the Benefits Case to Organizational Priorities:** Depending on the group's specific dynamics, customize the benefits case to align with their priorities. It might be beneficial to seek guidance from a coach who possesses insights into the group. Emphasize the significance of this endeavor in terms that resonate with them, and illustrate how the resulting output can be harnessed to deliver value for their business. Aligning with their priorities enhances the likelihood of a more receptive response to this effort.

Perform Prep Work

When you request information that is readily available, there’s a potential risk of giving the impression of squandering their time, leading to stakeholder frustration, and jeopardizing the entire initiative. Certain assets can detail many of the organization’s activities. Investing the necessary time in this prep work is essential to prevent entering sessions with a blank sheet of paper. It’s possible that a portion of this prep material might be outdated, which is fine as it will focus the interview sessions on making necessary updates. Demonstrating this level of preparation will be greatly appreciated by your stakeholders, showcasing your commitment to both their efficiency and time savings. The activities in this phase are outlined in Table 9-2.

Table 9-2. Perform Prep Work Activities

Activities	Outcome(s)
• Review Gathered Material: Carefully review all the gathered material, paying particular attention to verbs that indicate actions. If there’s a substantial amount of material, there’s a chance of identifying gaps or even conflicts among them. In such cases, prepare a list of clarifying questions for engagement with process owners. • Draft the Process Inventory: Utilize the collected material to draft the Process Inventory. Take note of areas where gaps exist, requiring further clarity from stakeholders.	• Draft Process Inventory

Best Practices

- **Engage a Coach:** The Process COE team leading this effort may not possess expertise in every organizational unit, potentially making it difficult to interpret the prep materials. Identify a coach within the organizational unit who can assist in navigating these documents and addressing any clarifying questions. Collaborating with a coach will yield substantial value from this step.

Standards

This is the initial step in creating the Process Inventory model for an organizational unit. It’s crucial to ensure compliance with modeling standards before presenting the draft to organizational unit stakeholders.

- **Organizational Structure Alignment:** The key to being comprehensively exhaustive in the Process Inventory is to construct it following the organizational structure. This should be the case in your initial draft as this will facilitate productive discussions when you are interviewing stakeholders.
- **Identify Process Owners:** Each identified process must have an owner. When crafting this draft, make an effort to pinpoint owners, with adherence to the organizational structure being crucial. Stakeholders in the subsequent phase can then validate or refine your draft.
- **Mutually Exclusive:** Each identified process should be uniquely defined and distinctly differentiated from others.
- **Identify All Unique Processes:** It's not uncommon for various areas within an organizational unit to carry out similar processes, such as project management and issue management. While the instinct might be to group them together under a common category in the Process Inventory, it's important to resist this inclination. Each of these seemingly similar processes may be executed and managed differently. Giving into this urge to group them could limit the potential of various use cases, including uncovering organizational inefficiencies, identifying overlaps, and effectively managing risks.
- **Determining the Process Level:** Establishing the appropriate level of process decomposition is more of an art than a precise science. Deciding when the Process Inventory has been adequately broken down and it's time to proceed with process modeling is a nuanced task. Process can be modeled at multiple levels if their activities follow a logical and continuous sequence. The Process COE team overseeing this effort must possess the expertise to discern the suitable level for initiating process modeling.
- **In Taxonomies, a Child Can Only Have One Parent:** This further underscores the point of uniqueness of processes. A single process cannot be mapped to multiple points in the Process Inventory taxonomy.

- **Use Verb + Noun Naming Convention:** Having consistent and clear naming is critical for any stakeholder to leverage this asset. This naming convention provides that as the verb specifies the action and the noun represents the entity on which the action is performed on.
- **Avoid Vague Verbs:** The focus should be on avoiding verbs that are vague in process names, which don't clarify the action such as manage or support. For the upper levels of the taxonomy, these work fine, but for a process, it's hard to understand what is exactly happening.
- **Spell Out Uncommon Acronyms:** To achieve clarity for individuals unfamiliar with the process or its domain, spell out uncommon acronyms that are only familiar to those with expertise in the domain.
- **Capture Process Descriptions:** Ensure the process descriptions are concise, spanning approximately two to four sentences, to offer enhanced clarity regarding the process actions.
- **Decompose to No More Than 15:** When a parent in the taxonomy has more than 15 children, the decomposition becomes increasingly unreadable. In such cases, consider defining additional categories and appropriately grouping the processes or process levels within them.
- **Minimum Decomposition:** To decompose a top-level node of the Process Inventory taxonomy to the next level, it's best practice to have a minimum of three at that next level. Decomposing 1:1 just means that you are renaming the top level.
- **External Vendors:** When processes are outsourced to an external vendor, it's still important to collect an inventory of *what* that vendor is performing as your organization is still accountable for those processes. Depending on the contract, you may not have the ability to perform detailed process modeling, but having an inventory of processes will enable proper oversight of the vendors' performance and compliance with your policies.

Conduct Interview Sessions

In this step of the process, interview sessions are conducted with owners within the organizational unit. These sessions initiate at the top of the organizational hierarchy, often starting with the organizational unit's leader, to establish the structure of the Process Inventory taxonomy for the organizational unit. Sequentially, iterative sessions are carried out at progressively lower levels, culminating in a comprehensive inventory of processes. This iterative approach serves to validate the process structure while also confirming the accurate identification of process owners.

During these sessions, drafts created in the previous step serve as the starting point. Process and process area owners actively participate, leveraging their expertise to refine the draft to become accurate and comprehensive.

Table 9-3. Conduct Interview Sessions Activities

Activities	Outcome(s)
• Set Up Working Sessions: Collaborate with leadership and a designated champion within the organizational unit to determine the most suitable meeting times. Typically lasting around one to two hours although the duration may vary based on the group's size and complexity. • Conduct Interviews: Focus on comprehensively inventorying the processes undertaken by the organization. If the need arises to delve deeper within the organization, confirm the involvement of relevant attendees for subsequent follow-up sessions. • Identify Opportunities for Improvement: Although not the primary focus of these sessions, it's possible that the team may recognize opportunities for enhancement. Formally capture these observations and present them to leadership during the final review as potential avenues for improvement. • Take Detailed Meeting Minutes: Capture comprehensive meeting minutes, record the session, and share them with the attendees after the discussion, expressing appreciation for their active participation.	• Process Inventory • Process Owner Names • Improvement Opportunities

Best Practices

- **Prioritize the Process Owner:** While inviting additional functional experts like technology and risk specialists to the sessions might be beneficial, it's crucial to emphasize the central role of the process owner. In cases of disagreement, the authority of the process owner should be recognized as final. This approach not only reinforces the significance of the process owner role but also prevents extended debates that could extend the effort's duration.
- **Generate Work Product in the Sessions:** The most efficient approach to compiling the inventory is to craft the work product during the session itself. This method allows attendees to rectify errors on the spot and avoids the need for subsequent offline exchanges, which inevitably leads to misinterpretations, to make corrections.
- **Cultivate Curiosity:** When engaging with stakeholders, they might not remember every aspect of their work. Use open-ended questions with sincere curiosity to prompt them. This approach is effective for uncovering valuable details necessary for a comprehensive representation.
- **Effective Challenge Resource(s):** Especially in situations where the organization deals with complex tasks, it's worth considering the participation of a subject matter expert(s). This expert(s) can attend the sessions to ask clarifying questions and fulfill the role of an effective challenger(s), leveraging their domain knowledge to prompt discussions and ensure accuracy.

Align Metadata and Perform Reviews

To obtain the final Process Owner attestation of the Process Inventory for an organizational unit, the final package needs to be refined with all required metadata and undergo a rigorous review to ensure compliance with documented standards. This step is crucial to ensuring that the package accurately represents the real-world environment and can be leveraged in various use cases for business value.

Table 9-4. Align Metadata and Perform Reviews Activities

Activities	Outcome(s)
• Confirm Process Descriptions: Ask process owners to review and confirm the written descriptions of each process. • Align Metadata: Identify and align the required metadata according to the documented process standards. This may involve discussions with additional functional stakeholders such as risk, technology, and others, as well as validation from process owners to ensure alignment. • Conduct Quality Reviews: This involves ensuring that the Process Inventory is clear and complies with documented standards. • Review with Process Owners: Conduct a review with process owners to validate the accuracy of the complete product. • Obtain Process Owner Attestation: Request process owners to provide formal attestation to validate the accuracy and completeness of the work product.	• Metadata Alignment • Quality Review Acceptance • Process Owner Attestation

Best Practices

- **Standards Checklist:** Standards for Process Inventory and process models should be documented and distributed broadly in the form of a checklist. This ensures that process team members have a clear understanding of what constitutes a quality work product and enables them to self-assess their work.
- **Levels of Reviews:** Typically, there are multiple levels of review. Initially, the individual creating the models should review their own work against the standards checklist. Subsequently, a peer review is conducted by a team member not involved in this specific effort. Finally, an independent review is carried out by a member of the Process COE control tower team. This multilevel review approach ensures high quality and reinforces the importance of quality throughout the team.
- **Review Workflow:** Utilizing a workflow, preferably available within the Business Process Analysis tool, is the preferred method for effectively organizing, tracking, and documenting both quality reviews and Process Owner attestations.

- **Leverage Metadata Libraries:** It's essential to use metadata from these libraries, where applicable, to maintain traceability to authoritative sources and enable impact analysis and reporting. Chapter 10 will explain how to set up metadata libraries in Business Process Analysis tools.
- **Types of Metadata:** At this stage, any applicable metadata can be aligned. Typically, metadata that doesn't require detailed process modeling is aligned. The following list, while not exhaustive, provides examples of metadata mapping that may be required.

Table 9-5. Sample Metadata to Align

Meta Data	Description
Process Description	A concise description consisting of two to four sentences, aimed at being comprehensible to individuals outside of this specific business area.
Process Owner	This is the designated individual who holds ownership of this process within the organizational unit. As the Process Inventory follows a hierarchical structure, the complete lineage of owners is traced throughout the taxonomy.
Product(s)	These are the product(s) that this process supports. In shared service organizations, it's not uncommon for a process to support multiple products. Additionally, certain support processes might not directly relate to any specific products.
Channel(s)	For customer-facing processes, especially those supported across various channels, this approach offers a valuable means of comprehending this relationship. This understanding enables the creation of a consistent omnichannel experience for customers.
Business Capability Model and Standard Process Model Alignment	This involves matching each process identified through the Process Inventory exercise to the corresponding capability or process identified in the enterprise's adopted model.
Risk and Controls	This typically requires the support of the first line of defense risk managers. Extracting risk and controls from the GRC or risk repository will position the team to conduct more robust risk assessments and design more effective controls.

- **Attestation Template:** Before engaging with a business area, make sure to clarify the expectations that you'll be asking them to confirm the accuracy and completeness of the information. Create a standardized template that will be used with all process owners and stored for future reference.

Conduct Readout with Leadership

This step is crucial within the approach, offering leadership the chance to review the final product for their organizational unit and provide any feedback before concluding this effort. It also serves as an excellent occasion to discuss how the Process Inventory can contribute to their future goals.

Table 9-6. Conduct Readout with Leadership Activities

Activities	Outcome(s)
• Prepare the Final Readout Packet: This should include the completed Process Inventory, the list of attestations collected, identified opportunities, and suggested next steps. • Conduct the Readout Session: Review the information packet and adjust based on their feedback. • Discuss Opportunities and Next Steps: Particularly highlight any high-priority areas the team should address in a subsequent effort.	• Final Read Out Packet • Organizational Unit Leader Attestation • Prioritize List of Opportunities and Next Steps

Best Practices

- **Gain Alignment Prior to the Meeting:** These read-outs go smoothly when all stakeholders within the organizational unit are aligned with the content of the final readout packet prior to the discussion with leadership. Take the time to review the packet with key stakeholders before the final discussion.

Process Models

Process models are a natural extension of Process Inventory as they depict the “how” behind the execution of inventoried processes. The key to understanding their relationship lies in recognizing that process models must adhere to the bounded context established in the Process Inventory. This means that each process model should exclusively represent a single process within the Process Inventory.

The creation of process models may occur immediately after developing the Process Inventory, or there might be a time gap between these steps. Regardless of the timing, it's crucial to grasp that the Process Inventory might evolve during the process modeling phase. Teams delving deeper into process understanding might identify the need for updates to the inventory of processes.

In some expansive organizations, the Process Inventory may encompass tens of thousands of processes. It's not feasible to expect every process will be documented with a process model. Prioritization of which processes to document is typically guided by the digital strategy, key use cases, and business demands. Regardless of how priorities are determined, maintaining the integrity between Process Inventory and process models remains paramount.

Construct Process Models

The goal is to achieve consistent documentation and maximize the efficiency of process owner engagement. Therefore, adopting an industrialized approach becomes crucial. This approach encompasses detailed standards, well-defined accountabilities, and multiple review checkpoints to ensure the creation of high-quality process models.

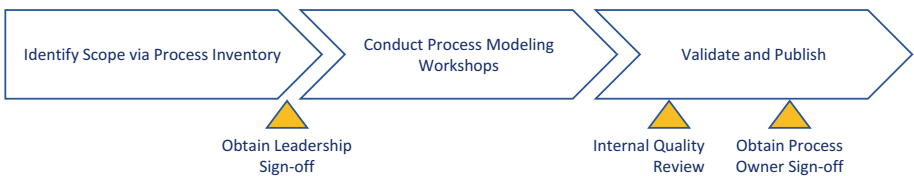


Figure 9-2. The Process Modeling Process

Identify Scope via Process Inventory

As mentioned earlier, this step upholds the integrity of the entire Process Inventory framework. This is crucial for maintaining end-to-end process modeling as there may be handoffs across processes, making the preservation of relationships and connections between events imperative.

However, its significance goes beyond that as it distinctly defines the scope of any process modeling effort. This enables engagement with relevant process owners and provides precision in estimating resource efforts and timeline durations.

Table 9-7. Identify Scope via Process Inventory Activities

Activities	Outcome(s)
• Match Request to Process Inventory: Engage with the requestor of the modeling effort to comprehend the processes they intend to have modeled. Correlate the requested processes with those identified in the Process Inventory. • Estimate Process Modeling Effort: The process modeling team should maintain an estimation tool. The scope of processes serves as input criteria that determine the total hours required from each involved resource, along with the estimated timeline. An in-depth plan should be the outcome of this step. • Identify Process Owners' Availability: Assess resource schedules and pinpoint the suitable timeframe for process modeling workshops. • Gain Acceptance on the Plan: Convene with the requestor and other key stakeholders to review the plan and gain buy-in for the effort.	• Identified Process Inventory Processes • Resource and Timeline Estimates • Agreed Upon Plan

Best Practices

- **Maintain Process Modeling Estimation Model:** The key to an industrialized process modeling capability is maintaining an estimation model which takes the details set of steps in the process and determines which resources are required and how many hours are needed for each. When applying the total process count in any scope, this will produce a total amount of hours. This model should be continuously analyzed and improved as the team learns from previous efforts.

Conduct Process Modeling Workshops

Process modeling workshops can be conducted to document either the current process or design a future state process as part of an initiative. Each scenario is distinct: current state workshops center on the identified process owner detailing the existing process, while future state process models may involve a broader set of stakeholders envisioning the path ahead. Irrespective of the context, the outcome of these sessions is to document the process, capturing the sequence of activities and the resources leveraged in execution.

Table 9-8. Conduct Process Modeling Workshops Activities

Activities	Outcome(s)
• Review Process Inventory: Validate with the process owner and additional stakeholders that the Process Inventory and the definitions of the processes in scope are accurate and up-to-date. • Walk Through the Process: Ask the process owner to explain the sequence of activities, starting from the initial event(s) and progressing to the ending event(s). • Document the Process: While the process owner is explaining the process, the process modeler should simultaneously model it during the session. • Identify and Map Metadata: This step involves documenting the specific technical systems that facilitate activities, data utilization, organizational and third-party actors, customer touchpoints, risks, controls, and more. Collaboration with additional functional stakeholders may be required to ensure alignment.	• Validated Process Inventory • Drafted Process Model • Identified and Mapped Metadata

Best Practices

- **Start with Process Boundaries:** Initiate the process by confirming the starting and ending events. This approach establishes the process's scope and directs the discussion toward the activities within that defined boundary.
- **Create Process Models in the Workshop:** Develop the process models directly during the workshop using the process modeling tool. This method is highly efficient, allowing the process owner to provide real-time feedback and eliminating the need to input the model into the tool at a later stage.
- **Invite Subject Matter Experts:** Extend invitations to experts in functional areas like technology, data, and risk. Their specialized insights can provide a deeper understanding of certain aspects that the process owner might not be familiar with.
- **Current State and Future State:** Recognize that both current state and future state processes may coexist for a particular process, especially in initiatives where a future deployment will alter the current process. If the current state process has already been documented, it serves as a valuable starting point for future state discussions.

Standards

Adhering to process modeling standards is crucial for documenting clear and consistent processes across the enterprise. Organizations need to document and maintain standards that are appropriate for their context. These standards should be distributed to all process modelers in the form of a checklist to ensure transparency and guide them in creating models that align with these rules. The standards listed in the following are not exhaustive but represent the types of rules to enforce:

- **Adherence to BPMN 2.0 Modeling Standards:** The BPMN 2.0 process modeling standard, created by the Object Management Group, maintains specifications. Many standards for creating good process models should be derived from this specification.
- **Is the Process Clear?** Process models shouldn't read like inside jokes. The test is, can someone unfamiliar with the domain read and understand the activities in the process model? If not, further edits are required to clarify it.
- **Alignment to Process Inventory:** The process model needs to maintain the integrity of the relationship with the Process Inventory. This means that it should fit within a single process and align with the definition given. Process Inventory may need updates, but that integrity must be maintained.
- **Event Naming:** I've seen many process models that start with an event named "start" and end with an event named "end," which is of negligible value. Event names need to be descriptive of the event with a noun plus a past tense verb as the naming standard.
- **Activities Documented with Verb + Noun Convention:** As each activity in a process can be detailed with a subprocess, then every activity needs to maintain the verb + noun naming standard. This also provides clarity to anyone reading the model.
- **No More Than 15 Activities:** To maintain readability, process models should not exceed 15 activities and should be printable on a standard sheet of paper. If more details are necessary than can be accommodated by this standard, subprocesses should be developed to offer additional levels of detail.

Validate and Publish

Process models, especially for critical processes, need to maintain a high level of quality to maximize the benefits for the organization. This phase is an important step to validate that a high level of quality is achieved.

Table 9-9. Validate and Publish Activities

Activities	Outcome(s)
• Conduct Peer Review: This is performed by a member of the process modeling team who wasn't part of this specific effort. The quality checklist is leveraged to ensure the model complies with all standards. • Conduct Final Quality Review: A member of the Process COE control tower team will perform a second-level review to catch issues not found in the peer review. • Review with Process Owners: Conduct a review with process owners to validate the accuracy of the complete product. • Obtain Process Owner Attestation: Request process owners to provide formal attestation to validate the accuracy and completeness of the work product.	• Process model(s) which complies with standards • Process Owner attestation

Best Practice

- **Perform Internal Review Prior to Process Owner Attestation:** The process model presented to the process owner should comply with all standards. This will avoid having to ask the process owner to attest additional times if quality issues are found. This will save time and frustration for the process owner.

Key Takeaways

- **Modeling the environment** through Process Inventory and process models is the key to maximizing digital technologies, achieving operational excellence, and delivering competitive business value.
- **Guiding principles** are essential to ensure that your models offer optimal value to the organization by adhering to standards of quality, clarity, and consistency.

- **An industrialized approach** is required to create high-quality modeling, which adheres to consistent standards and makes efficient use of stakeholders' time.
- **Maintaining the relationship** between Process Inventory and process models is critical for delivering on the potential of this framework.
- **Delivering quality models** requires detailed standards and a rigorous review approach.

In the next chapter, we will explore commercially available Business Process Analysis tools and their essential features required to support the modeling of the process environment, as well as the management of metadata.

Modeling Platform and Data Infrastructure

In this chapter, we will explore the importance of Business Process Analysis platforms in managing the Process Inventory framework, associated models, and supporting metadata. You will gain an appreciation for its role in providing alignment across the organization to deliver digital transformation and operational excellence goals.

We will cover a select set of platforms available in the marketplace. We will delve into key features and considerations for implementing and gaining value from Business Process Analysis platforms.

Lastly, we will cover a methodical approach for selecting the right tool for an organization and performing the initial setup.

FedEx Dataworks: Leveraging Operational Data for Business Performance

In the fast-paced world of business, operational data has emerged as a potent catalyst for driving enhanced performance. FedEx demonstrates the profound impact of leveraging operational data to fuel business success. By harnessing the power of analytics, artificial intelligence (AI), and machine learning, FedEx has elevated its processes, redefined operational excellence, helped its customers absorb shocks to markets and the supply chain, and gained a significant competitive advantage.

A pivotal philosophy ingrained in FedEx's DNA is the vision that the company's founder, Fred Smith, recognized: the idea that information about a package is just as vital as the package itself [1]. This belief laid the foundation for FedEx's strategic approach to data-driven decision-making. At the heart of this is the ability to transform a staggering 18 million daily shipments across 220 countries into actionable insights.

To further this philosophy, the company built the Dataworks platform, which seamlessly integrates the company's physical and digital networks. This platform is a centralized hub that provides connective tissue to integrate all available data sources within FedEx's global network. They leverage this to track every aspect of a package's movements, including real-time data such as delivery route, environmental conditions, weather, customer delivery locations, and delivery date. This enhances supply chain visibility, enables the identification of deviations from planned paths, improves predictability, and facilitates data-driven decision-making, benefiting high-value and time-sensitive shipments.

This insight has led them to offer FedEx Surround to their customers. Surround is designed for businesses to digitize their supply chains by leveraging data to drive breakthrough real-time inventory tracking and logistics management, which is a digital representation of the physical world [3]. This solution integrates IoT data through SenseAware ID [4], a small Bluetooth tracking sensor that can be placed on a package. This enables FedEx to intervene if a package is at risk of not arriving on time. For example, a team can scoop up the package at a sorting center and place it in a van to leave immediately. From origin to destination, there are multiple intervention points.

To be successful, they designed this platform to be free of operational silos, where data, models, and digital capabilities could be reused across multiple use cases. This reinforces core capabilities, drives innovation in processes and features, accelerates time to market, and ensures accurate decision-making. This centralized hub fosters cross-functional collaborations and the seamless exchange of data.

As organizations navigate the intricacies of modern business, the FedEx Dataworks journey serves as an illustration, highlighting the power of managing and exploiting operational data in shaping a new era of innovation and driving differentiated business performance.

Business Process Analysis Platform Market

It cannot be overstated that to realize the potential of the Process Inventory framework to drive digital transformation and achieve operational excellence, a platform is crucial for modeling the process environment and effectively managing all associated operational metadata in a scalable and sustainable manner. To effectively support the digital strategy, this platform must encompass extensive modeling capabilities, facilitate data integration across numerous authoritative sources of truth, offer capabilities to analyze data to optimize processes or address critical issues, and support accountability through an ownership model, among other essential features.

Fortunately, a mature vendor market of platforms, known as Business Process Analysis (BPA), that provides the necessary capabilities exists. These platforms can successfully implement the Process Inventory framework and harness the benefits of alignment.

In the following, I will present a select set of companies and products within this market, accompanied by a brief description of how each vendor defines their platform. While this list is not exhaustive, it represents vendors and products with which I have some familiarity. I encourage each organization to conduct its own due diligence based on its specific needs and context.

Having worked with several of these platforms in large programs, I can attest that many of their capabilities are impressive, continue to evolve rapidly, and align well with the requirements of implementing the Process Inventory framework.

Table 10-1. Select Vendors and Vendor Platforms in the Business Process Analysis Market

Vendor/Platform	Vendor's Description
Avolution—Abacus [5]	Connect your IT, business processes, and business strategy. Cloud-based data, dashboards, and diagrams support teamwork across locations and departments.
Bizzdesign Horizon [6]	The business and change platform to design an agile Enterprise Architecture
iGrafx Process360 [7]	The products and capabilities of Process360 Live combine next-generation operational intelligence, automation, and process design to transform your business.
MEGA HOPEX [8]	Build a digital representation of your enterprise by connecting business, IT, data, and risk perspectives in a single platform to share a single source of truth. Derive actionable insights and collaborate with stakeholders to align your company's business objectives and demonstrate the immediate business value of your projects. Seamlessly integrate the HOPEX Platform into your digital ecosystem for faster time to value.
Orbus OrbusInfinity [9]	Our business-centric platform creates a holistic view of your applications, systems, and data, so you can accelerate decision-making, reduce spend, mitigate risks, and stay resilient in the face of disruption.
SAP Signavio Process Transformation Suite [10]	With our SAP Signavio Process Transformation Suite, organizations can understand, proactively manage, and optimize their business processes to achieve the agility needed for successful change.
Software AG ARIS [11]	ARIS improves efficiency, saves time and money, and makes processes more sustainable. It offers true business process transformation via tightly managed and controlled change. With ARIS, you can create, manage, and analyze your end-to-end business processes—the key to operational excellence that gives stakeholders the tools and insights they need to make better decisions every day.

Key Business Process Analysis Features

When evaluating and implementing a Business Process Analysis platform, it's important to develop requirements that align with the context of your organization. This alignment should extend from both the digital strategy and the Process Inventory strategy, ensuring that the platform can robustly support these overarching visions. Implementing a platform requires a significant commitment of effort and resources, making it essential to weigh short-term goals as well as long-term needs.

In this section, I will delve into some of the key features of this platform and highlight their critical role in supporting the Process Inventory vision.

Modeling Capabilities

The most crucial features of these platforms are their ability to create and maintain models that comprehensively represent the organization from diverse perspectives, integrating them to support analysis from multiple viewpoints. Additionally, these platforms must construct models in a manner that facilitates value extraction through analysis.

General Modeling Capabilities

- **Object Orientation:** Real-world concepts or entities, such as processes, systems, and events, are represented as objects. These objects are self-contained units that encapsulate attribute data and behaviors. These objects are reused across models, enabling analysis and reporting on the use of these concepts or entities.
- **Support Multiple Viewpoints:** These platforms need to serve the needs of a diverse set of stakeholders such as leadership, business, technology, data, and risk. By providing customizable views tailored to specific roles, such as strategy, process, technology, and data, the platform ensures that each stakeholder can focus on the information most relevant to their responsibilities. This capability enhances collaboration while anchoring these views to the common language of Process Inventory.
- **Integration Across Model Types:** Linkages across various types of models, anchored to Process Inventory, are important for viewing the business from multiple angles and identifying impacts at various levels of granularity. This ensures consistency and alignment and maintains a comprehensive view of the organization. Integrated models facilitate the identification of dependencies and potential impacts, supporting holistic process optimization.
- **Version Control:** This capability enables organizations to maintain a historical record of changes made to models, ensuring traceability and accountability. Stakeholders can create, compare, and revert to different versions of process models. This is also important for maintaining the current state as well as one or more future states when the organization is implementing change. This feature safeguards against unintended modifications, supports regulatory compliance, and enables teams to analyze the impact of changes before implementation.

- **Import Existing Models:** Organizations often have existing repositories of models created in various tools such as Microsoft Visio. This feature allows seamless migration and integration of these models into the chosen platform, avoiding the need to recreate them from scratch. By preserving existing investments, organizations can save significant time and effort in modeling the organization comprehensively.
- **Collaboration:** The platform should facilitate real-time collaboration among multiple stakeholders, allowing them to co-create, review, and edit process models simultaneously. Collaborative tools, such as commenting, annotation, and discussion threads, enable stakeholders to provide feedback and contribute insights directly within the models. This feature enhances cross-functional collaboration, reduces silos, and ensures that process analysis benefits from the collective knowledge and expertise of the entire team.
- **Customization:** These platforms typically support many modeling standards and will have numerous preloaded frameworks. They should also include the ability to customize the platform to an organization's specific needs. This includes the ability to create user-defined model types or elements which is crucial if an organization has unique characteristics.

Model Types Supported

- **Strategy Model:** These models capture the high-level strategic goals and objectives of the organization. They outline the long-term vision and the path to achieve it, serving as a guiding framework for decision-making. Strategy models can be mapped to processes and capabilities to aid in prioritization and ensure that initiatives are aligned with the organization's overarching strategy. The platform should enable the visualization, analysis, and communication of strategy models to facilitate informed decision-making and ensure alignment across the organization.
- **Customer Journeys:** Customer journeys play a pivotal role in understanding and enhancing the customer experience. The platform should enable teams to create customer personas and map out the journey and the

various touchpoints and interactions that customers have with the business as they leverage a product or a service. These models illustrate the end-to-end customer experience, helping organizations identify pain points, opportunities for improvement, and areas to enhance customer satisfaction.

- **Value Streams:** Value streams demonstrate the set of end-to-end actions that take place to add value to the customer from the initial request through the realization of value by the customer. The platform should enable teams to define and map value streams, encompassing all the activities, processes, and interactions involved in these end-to-end processes.
- **Process Inventory:** The backbone of this framework is the Process Inventory taxonomy. The platform should offer the capability to create, manage, and maintain this taxonomy for the current state and for future state changes. This model serves as the storage structure for all process models within the organization, facilitating process documentation, analysis, and optimization. The platform should support the construction of this at multiple levels of granularity and capture the aligned process owners.
- **Capability Model:** Capability models represent high-level views of a business from the perspective of the capabilities they perform. These models are largely independent of organizational structure. When paired with the organizational structure-aligned Process Inventory, they provide business leaders with different viewpoints from which to analyze their business. When used for planning purposes, these models depict the critical capabilities that enable the organization to deliver value to its stakeholders and achieve its goals. This enables leaders to make informed choices about where to focus investments, streamline operations, and drive innovation across organizational boundaries.
- **Business Process Model:** Business process models offer a comprehensive depiction of “how” individual processes are performed, illustrating the sequence of activities, roles, and interactions involved. Business Process Analysis platforms should provide a range of tools to facilitate the creation, visualization, and analysis of business process models, typically adhering to standard

modeling notations like BPMN 2.0. It's noteworthy that in certain scenarios, BPMN 2.0 diagrams can be seamlessly imported into workflow tools, which then play a crucial role in generating workflow code needed in the execution of long-running processes.

- **Business Rule Modeling:** Business rules play a pivotal role in maintaining consistency, compliance, and informed decision-making across an organization. A proficient Business Process Analysis platform should offer robust support for Business Rule Modeling, utilizing the widely adopted Decision Model and Notation (DMN) standard. DMN is a standardized approach for defining and modeling repeatable decisions, with the advantage of being easily comprehensible by both business and IT stakeholders. Typically integrated into BPMN processes, DMN enables the creation of decision logic, rule tables, and decision diagrams, effectively aligning business rules with the overall process flow and enhancing the transparency and efficiency of decision-making processes.
- **System Sequence Diagram:** As discussed in Chapter 6, system sequence diagrams illustrate the information flow and interactions among systems. When integrated with business process models, they provide a comprehensive view of the systems that support process activities. By linking system sequence diagrams to business process diagrams, organizations can pinpoint integration nodes, detect possible failure points, and uncover opportunities to enhance system performance. While this feature is present in select Business Process Analysis platforms, it may also necessitate integration with an Enterprise Architecture tool.

Process Mining

Many Business Process Analysis platforms have integrated process mining capabilities, which are instrumental in leveraging available system data to gain insights into the execution of processes and identifying opportunities for improvement. While there are stand-alone process mining platforms, the insights derived become more valuable when aligned with the organization's models. These capabilities encompass several key features:

- **Process Discovery:** By analyzing event data from system logs, this functionality creates process models that illustrate the sequence of activities, decisions, and interactions. It enables organizations to validate processes documented from stakeholder interviews with real-world data. This can help identify potential variations, inefficiencies, and bottlenecks, paving the way for targeted improvements and optimizations. As mentioned in Chapter 2, to maximize the effectiveness of this feature, the output should be harmonized with the Process Inventory, allowing for the alignment of metadata and benefits beyond process optimization.
- **Conformance Checking:** Conformance checking is a critical feature that focuses on comparing discovered or actual process instances with predefined or expected process models. By doing so, organizations can assess the level of alignment between designed processes and their real-world execution. This process highlights discrepancies, deviations, or noncompliance instances, which could indicate process inefficiencies or unacceptable risks.
- **Performance Analysis:** This functionality enables organizations to quantify the performance of processes by analyzing metrics such as process cycle time, throughput, bottlenecks, and resource utilization. By assessing these performance indicators, stakeholders gain insights into the efficiency and effectiveness of their processes, which points them to specific opportunities for process optimization. This data-driven approach enables organizations to make informed decisions about process enhancements, resource allocation, and change strategies.

Data Management Capabilities

The Business Process Analysis platform, especially when equipped with extensive process models and metadata, serves as a pivotal authoritative source of data that describes an organization's entire operational landscape. Given this significance, I referred to it as the Integrated Operational Repository. The implications of this status necessitate an organization's commitment to maintaining the quality and accuracy of this data with rigor to lock its potential benefits. As a result, the Business Process Analysis platform requires sophisticated functionality to effectively manage this data, ensuring alignment with organizational goals and driving insightful decision-making:

- Import Metadata:** The platform will need to import data from key authoritative sources of information to integrate with the models built on the platform. The data to import should be aligned to the use cases and benefits that are defined in the Process Inventory strategy. The information imported should be limited to the data required for modeling, analysis, and reporting on this platform. This typically includes ID information along with a few more descriptive fields. Additional details can be retrieved from the source when needed. There will be an initial load of data and periodic refreshes as the original source undergoes changes. During subsequent imports, conflicts will inevitably occur, requiring workflow and manual work to address these conflicts. Figure 10-1 depicts the importing of data in a conceptual architecture diagram.

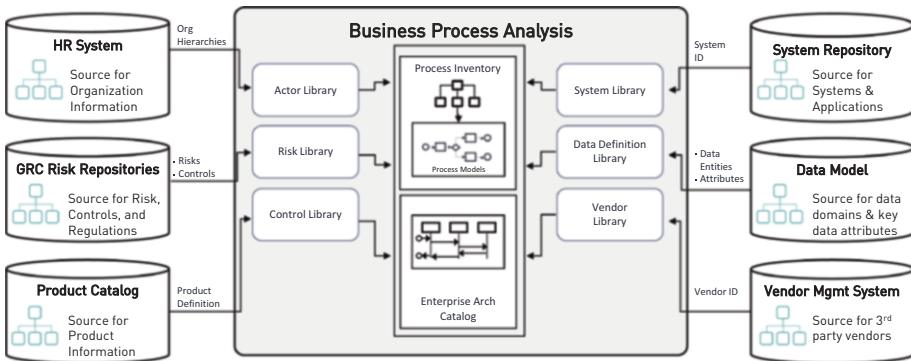


Figure 10-1. Conceptual BPA Architecture

- Manage Data Libraries:** The platform will have to include the ability to establish libraries with this imported data and other modeled data, such as capability models. Teams creating models will need access to this metadata to integrate into their models. This establishes the relationship from the appropriate points in the digital models that represent the real world to data in disparate authoritative sources of data, which is so critical for alignment.
- Export Data:** As this platform becomes the authoritative source for an organization's operating information and establishes a common language for alignment, other platforms across the organization will require

this data. Figure 10-2 depicts this sharing of data. The most prominent requirement will be exporting Process Inventory data to platforms that generate data, such as transaction processing systems, and issue management, to ensure accurate categorization at the source. External business intelligence tools can also serve as another destination for performing additional reporting and analysis on this data.

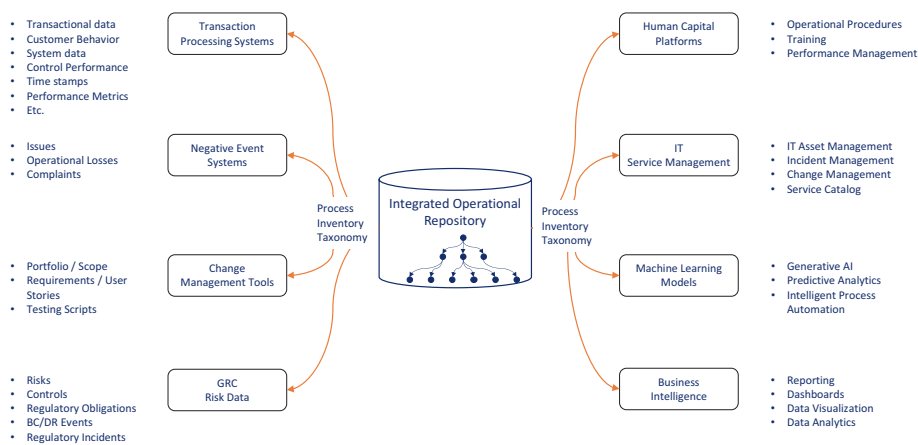


Figure 10-2. Publishing Process Inventory Taxonomy

- **Manage Data Quality:** Recognizing the pivotal role of data in driving accurate insights and informed decision-making, the platform should incorporate robust functionalities for effective data quality management. Central to this capability is the provision of visualizations, metrics, and reports that vividly portray the current state of data quality. These tools empower administrators to enforce data quality rules that align with organizational standards. This includes identifying and rectifying inconsistencies, anomalies, missing values, and potential quality issues present within the data. Moreover, the platform should empower administrators with tools to address inaccuracies, rectify duplicates, and resolve inconsistencies, thereby facilitating the sustained improvement of data accuracy and reliability.

Process Analysis

The comprehensiveness of the operational data this platform will store means that there are a tremendous number of insights to be mined which can be leveraged for continuous improvement, optimization of processes, and driving greater business performance. To capture this benefit, the Business Process Analysis platforms need to offer a suite of comprehensive process analysis features. In this section, we delve into the essential analytical functionalities that a robust BPA platform should provide:

- **Dashboards and Visualization:** Dashboard and visualization capabilities allow users to create interactive and customizable dashboards that display key process performance metrics and KPIs. These dashboards provide an overview of process health, trends, and patterns with ease. The tool should offer drill-down capabilities, enabling users to explore data at different levels of granularity and gain deeper insights into process performance.
- **Heatmaps:** This functionality enables users to visualize the status of various topics relative to the process environment using color-coded representations. Examples of heatmaps applied to process areas include strategic importance, performance, risk management, application redundancy, and more. These views focus leaders on important areas to focus on and outliers to address, facilitating data-driven decision-making and ensuring resources are allocated appropriately.
- **Impact Analysis:** This powerful feature enables users to explore the relationships between entities in the environment and assess the potential consequences of changes, issues, risks, regulations, and more. Users can model different scenarios and simulate changes to understand how they will affect other processes, systems, and stakeholders. Focusing on change management, this enables program managers to identify each resource impacted by a change initiative, resulting in comprehensive requirements, better engagement of resources, and more accurate estimates of cost and timeline. This helps organizations make decisions with comprehensive information and avoid unintended consequences.
- **Process Simulation:** Process simulation capabilities enable users to create and analyze various process scenarios with different variables, resources, and conditions to understand how they influence process performance. By running simulations, users can identify

opportunities for improvement, optimize resource allocation, and predict the outcomes of process changes. The platform should offer user-friendly modeling interfaces and provide detailed simulation results to guide decision-making.

- **Process Intelligence:** Process intelligence enables users to conduct in-depth analysis of process data. This includes identifying patterns, trends, and hidden opportunities for optimization. This feature leverages advanced analytics capabilities, such as process mining, text mining, and machine learning, to extract valuable insights from process data.
- **Root Cause Analysis:** When an issue occurs in the production or test environment, coordinating analysis across a group of stakeholders to identify the root cause can be challenging. The information within the Business Process Analysis platform, combined with root cause analysis features, allows users to analyze event logs and identify underlying causes of defects and inefficiencies.
- **Process Optimization:** Process optimization features provide recommendations for improving process efficiency and effectiveness. By analyzing process data, the tool can suggest changes, such as reordering tasks, optimizing resource allocation, or eliminating redundant steps. Users can explore optimization scenarios and assess the potential impact of changes before implementation, ensuring that process improvements align with organizational goals and strategic objectives.

Reporting

Robust reporting capabilities are essential to provide stakeholders with valuable insights into the process environment and identify trends and key metrics. These capabilities empower leaders and other stakeholders with valuable insights:

- **Out-of-the-Box Reporting:** These platforms should offer a selection of out-of-the-box reporting options that provide immediate insights into process performance and operational trends. These predefined reports are designed to address common reporting needs and offer a quick overview of processes without requiring extensive customization.

- **Custom Reports:** In addition to out-of-the-box reporting, users need the ability to create custom reports tailored to their specific requirements. This feature allows users to design reports that highlight specific key performance indicators (KPIs), performance metrics, or process aspects that align with their unique use cases and business objectives to drive informed decisions.

Governance

Just as data management capabilities are critical for ensuring the quality of data in the Business Process Analysis platform, robust governance is needed to maintain the accuracy, consistency, and compliance of the models, especially the Process Inventory and process models, and associated metadata. These features are key to effective model management, providing the tools and mechanisms needed to rigorously maintain a controlled environment:

- **Quality Assurance:** Quality assurance is a crucial governance feature that ensures the accuracy, consistency, and reliability of models by performing automated checks to validate and verify that models conform to defined standards and best practices. These automated quality checks should occur in conjunction with human reviews.
- **Administration, Roles, and Entitlements:** Effective administration, role management, and entitlements are integral to maintaining a controlled and secure environment for the assets stored in this platform. At a minimum, there should be three distinct roles: viewer, contributor, and administrator. Viewers have read-only access and are typically granted to those who consume models, such as business and other functional stakeholders. Contributors are the teams responsible for creating, updating, and managing models. Administrators, typically part of the core Process Center of Excellence (COE) organization, oversee and maintain the platform. The platform should allow roles and permissions to be tailored to different deployment environments. Administrators can define user access levels, privileges, and responsibilities that align with their specific needs. This ensures that the right individuals have the appropriate access to support collaboration, data integrity, and compliance.

- **Environment Management:** Environment management involves orchestrating and overseeing various deployment environments, each serving distinct purposes within the process life cycle. These environments, which include sandbox, development, test, and production stages, are crucial for validating, refining, and executing processes before they are fully deployed. The specific objectives of each environment and the criteria for transitioning between them should be clearly defined. For instance, the sandbox environment should be relatively open for modelers to explore concepts, while the production environment should be reserved for models that have undergone thorough quality reviews and attestations and accurately represent the current operational state. Models for future state processes, created through a change initiative, should only be promoted to the main production environment once the change has been implemented. Effective environment management ensures that stakeholders have an environment to leverage that accurately reflects the real world and adheres to high-quality standards. This approach not only safeguards operational integrity but also supports the separation of duties and minimizes disruptions during the testing and optimization phases.
- **Workflow Management:** Workflow management is a governance feature that facilitates the coordination and monitoring of tasks and activities within quality assurance and attestation processes, which often entail multiple steps and involve numerous stakeholders. This feature encompasses automated email notifications sent to process participants and dashboards that offer leaders real-time visibility into the status of approval requests.
- **Audit Trail:** These platforms need to maintain an audit trail which captures changes to models across different environments but also a detailed list of quality approvers and owner attestations. This is critical in maintaining accountability and tracing approvals if quality issues are raised that must be traced to the source to perform root cause analysis.

- **Risk Management:** Some Business Process Analysis platforms integrate governance, risk management, and compliance (GRC) functionality, which enables organizations to track their risks and regulatory obligations and ensures the presence of effective controls. While integration with an external GRC tool can suffice, this feature can be valuable if your organization lacks or requires an upgrade to this risk management functionality.

User Experience

The experience these platforms provide to your users cannot be overlooked. If a platform offers superior functionality, but users struggle to use it, then broad adoption will suffer:

- **Persona-Centric User Experiences:** The requirements for using the platform will vary greatly across the roles of viewer, contributor, and administrator. The population of contributors and administrators will be relatively low, and their needs will be robust, necessitating a more technical experience that should be adequately supported by training. The population of viewers will be larger, encompassing numerous stakeholders within the organization who will access the model. Therefore, extensive training isn't feasible, making it crucial to provide a streamlined experience tailored to their specific needs. In my experience, I've observed one organization switch their platform to another vendor primarily due to the perceived subpar experience.
- **Mobile Access:** In today's dynamic business environment, enabling mobile access is imperative for maintaining productivity and connectivity on the go. Mobile access allows users to view models and data using their smartphones and tablets to review process documentation, participate in approval workflows, or access key insights.
- **User Training:** To ensure that users can effectively leverage the capabilities of the platform, integrated user training modules are critical. User training should provide guided tutorials, documentation, and best practice guidelines to familiarize users with the tool's functionalities. By offering resources within the platform, users can quickly learn how to navigate, create, and analyze process models, ultimately enhancing their proficiency and confidence.

Business Process Analysis Platform Selection and Setup

Selecting and implementing the right Business Process Analysis platform is a significant activity especially if they are committed to pursuing a Process Inventory strategy to achieve an ambitious digital and business vision. Leveraging a methodical approach to documenting the organization's short-term and long-term needs and performing a rigorous platform evaluation is crucial in selecting the right tool that will set up the program for success. Figure 10-3 illustrates a high-level approach, which I'll delve into to highlight key steps, best practices, and considerations needed to succeed.

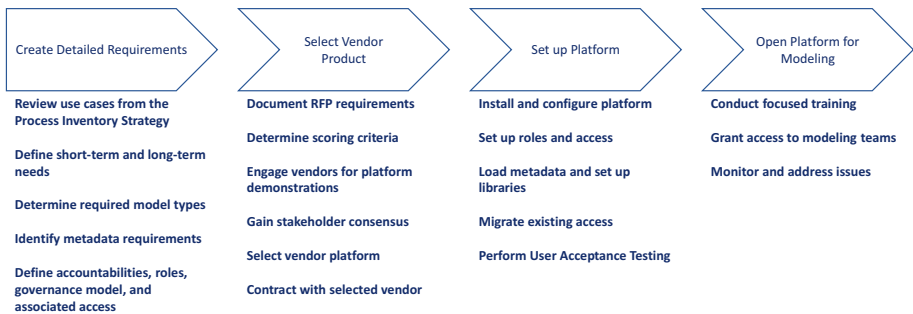


Figure 10-3. Platform Selection and Setup

Create Details Requirements

Creating detailed requirements for the chosen Business Process Analysis platform is a pivotal phase in ensuring alignment with the organization's digital strategy and the value-generating use cases identified through the Process Inventory strategy. This comprehensive set of requirements must encompass both immediate short-term objectives and a forward-looking vision for the platform's future needs. While my previous discussions have primarily focused on functional features, it's essential to recognize that technical considerations such as hosting options (vendor hosted cloud, internally hosted cloud, on-premises, etc.), data integration, and security also play a vital role in shaping these requirements.

The formulation of use cases, which articulate how the platform and models will contribute tangible business value, holds the key to determining the required model types and their structure. It even influences the specific data elements to be incorporated within the models. Use case requirements in the platform must be harmonized with the modeling standards and checklists crafted by the Process COE team to ensure a cohesive approach.

The process of defining metadata requirements involves active dialogue with the custodians of authoritative data sources. This interaction helps in comprehending the available data elements and gauging their quality. When encountering data of subpar quality, a critical decision must be made whether to omit such metadata even if use cases suggest their necessity. Such exclusions can be justified by the potential to maintain platform integrity and quality perceptions. Additionally, the collection of technical specifics is essential to shape data integration processes and routines for data quality validation.

Ultimately, the creation of an operating model design is important to outline the individuals responsible for model creation, utilization, and platform administration. This delineation significantly contributes to shaping access roles, entitlements, and privileges, thus establishing a robust governance framework. Through these meticulous steps, a well-defined set of requirements is forged, serving as the cornerstone for selecting and implementing the most suitable platform.

Select Vendor Product

Selecting the ideal vendor platform that aligns with your organization's unique needs requires engaging potential vendors to delve into their capabilities and future product vision. To initiate this process, crafting a well-defined request for proposal (RFP) is imperative. The RFP serves as the blueprint for your requirements, which is then distributed to potential vendors. This document guides vendors to showcase how their features fit with your needs.

Inviting vendors to present their platforms through product demonstrations is the subsequent step. This allows the evaluation team the opportunity to assess platform capabilities, gauge user experience, and seek clarifications directly from the vendors. A recommended practice is to provide vendors with high-priority use cases, which provide insights into the platform's functionality within scenarios integral to the organization's goals.

The evaluation process should be underpinned by well-documented selection criteria and a scoring mechanism. This ensures an objective assessment, enabling the identification of the best-fitting platform for your organization's needs. Given that this platform will serve various user groups, it's crucial to review recommendations with key stakeholder groups to address any concerns and gain consensus before proceeding with a selection.

While prioritizing features aligned with the Process Inventory strategy is vital, it's crucial to consider additional factors such as cost, vendor viability, and support during the selection process. Additionally, most vendors extend professional services to aid in platform installation and setup, a valuable resource particularly when the implementation team lacks detailed experience with the platform.

Setup Platform

Setting up the platform requires close collaboration with IT infrastructure teams for the installation and technical configurations. The administrator should oversee configurations that support the required model types and metadata libraries. The process includes loading libraries from their respective authoritative sources and importing existing models from previous tools to populate the platform with content.

Lastly, it's crucial to identify a subset of users who can test functionalities such as model creation, analysis, and report generation. This testing phase allows for the identification and resolution of any issues before deploying the platform to a broader user base.

Open Platform for Modeling

Before the full release, arrange training sessions that focus on key features and functionalities for a selected group of initial users and stakeholders. Once access is granted, administrators should be prepared to offer ongoing support to assist users in addressing issues and navigating their tasks.

Key Takeaways

- **Business Process Analysis platforms are critical** to manage the Process Inventory framework, associated models, and supporting metadata to deliver digital transformation and operational excellence goals.
- **An established market exists** that contains vendor platforms that have mature features that can enable success with the Process Inventory framework.
- **A methodical approach** is needed to select the right platform, involving documenting requirements, evaluating platform capabilities, and setting up the platform for modeling.

CHAPTER 1

Why Optimize?

This chapter covers why optimization is an important part of database development. You will learn the differences between declarative languages, like SQL, and imperative languages, like Java, which may be more familiar, and how these differences affect programming style. We also demonstrate that optimization applies not only to database queries but also to database design and application architecture.

What Do We Mean by Optimization?

While the majority of people think about database optimization as choosing the best system configuration parameters, in the context of this book, optimization means any transformation that improves system performance. This definition is purposely generic, since we want to emphasize that optimization is not a separate development phase. Quite often, database developers try to “just make it work” first and optimize later. We do not think that this approach is productive. Writing a query without having any idea of how long it will take to run creates a problem that could have been avoided altogether by writing it correctly way from the start. We hope that by the time you finish this book, you’ll be prepared to optimize in precisely this fashion: as an integrated part of query development.

We will demonstrate some specific techniques; however, the most important thing is to understand how a database engine processes a query and how a query planner decides what execution path to choose. When we teach optimization in a classroom setting, we often say, “Think like a database!” Look at your query from the point of view of a database engine, and imagine what it has to do to execute that query; imagine that you have to do it yourself instead of the database engine doing it for you. By thinking about the scope of work, you can avoid imposing suboptimal execution plans. This is discussed in more detail in subsequent chapters.

If you practice “thinking like a database” long enough, it will become a natural way of thinking, and you will be able to write queries correctly right away, often without the need for future optimization.

Why It Is Difficult: Imperative and Declarative

Why isn’t it enough to write a SQL statement that returns a correct result? That’s what we expect when we write application code. Why is it different in SQL, and why is it that two queries that yield the same result may drastically differ in execution time? The underlying source of the problem is that SQL is a *declarative language*. That means that when we write a SQL statement, we describe the result we want to get, but we do not specify *how* that result should be obtained. By contrast, in an *imperative language*, we specify *what* to do to obtain a desired result—that is, the sequence of steps that should be executed.

As discussed in Chapter 2, the *database optimizer* chooses the best way of doing it. What is best is determined by many different factors, such as storage structures, indexes, and data statistics.

Let’s look at a simple example: find the flights, including their departure and arrival airports, that are scheduled to arrive on October 14, 2023. Now, let’s consider the queries in Listings 1-1 and 1-2.

Listing 1-1. A query selecting flights with the comparison operators

```
SELECT
    flight_id
    ,departure_airport
    ,arrival_airport
FROM flight
WHERE scheduled_arrival >='2023-10-14'
    AND scheduled_arrival <'2023-10-15';
```

Listing 1-2. A query selecting flights by casting to date

```
SELECT
    flight_id
    ,departure_airport
```

```

    ,arrival_airport
FROM flight
WHERE scheduled_arrival ::date='2023-10-14' ;

```

These two queries look almost identical and should yield identical results. However, the execution time will be different because the work done by the database engine will be different. In Chapter 5, we will explain why this happens and how to choose the best query from a performance standpoint.

Thinking imperatively is natural for humans. Generally, when we think about accomplishing a task, we think about the steps that we need to take. Similarly, when we think about a complex query, we think about the sequence of conditions we need to apply to achieve the desired result. However, if we force the database engine to follow this sequence strictly, the result might not be optimal.

For example, let's try to find out how many people with frequent flyer level 4 fly out of Chicago for Independence Day. If at the first step you wanted to select all frequent flyers with level 4, you might write something like this:

```
SELECT * FROM frequent_flyer WHERE level =4;
```

Now that you have found all the records with level=4, you can use them to find the corresponding accounts. The query will look like this:

```

SELECT * FROM account WHERE frequent_flyer_id IN (
    SELECT frequent_flyer_id FROM frequent_flyer WHERE level =4
);

```

Great, you found accounts. On the next step, you want to find all bookings made by these accounts:

```

WITH level4 AS (SELECT * FROM account WHERE
frequent_flyer_id IN (
    SELECT frequent_flyer_id FROM frequent_flyer WHERE level =4
)
SELECT * FROM booking WHERE account_id IN
(SELECT account_id FROM level4);

```

But that's not it! Remember, the original goal was to find frequent flyers who traveled to Chicago on July 3. Continuing to build the query imperatively, we can come up with the code presented in Listing 1-3.

Listing 1-3. Imperatively constructed query

```

WITH bk AS (
  WITH level4 AS (SELECT *
                  FROM account
                  WHERE frequent_flyer_id IN (
                      SELECT frequent_flyer_id
                      FROM frequent_flyer
                      WHERE level =4
                  )
                )
    SELECT * FROM booking WHERE account_id IN
      (SELECT account_id FROM level4
      )
  )
SELECT * FROM bk WHERE bk.booking_id IN
  (SELECT booking_id FROM booking_leg WHERE
    leg_num=1 AND is_returning IS false
    AND flight_id IN (
      SELECT flight_id
      FROM flight
      WHERE departure_airport IN ('ORD', 'MDW')
      AND scheduled_departure:: DATE='2023-07-04')
  )

```

Now, imagine that in addition, you need to calculate the actual number of travelers. One way to do this would be with the query in [Listing 1-4](#).

Listing 1-4. Calculating a total number of passengers

```

WITH bk_chi AS (
  WITH bk AS (
    WITH level4 AS (SELECT *
                    FROM account
                    WHERE frequent_flyer_id IN (
                        SELECT frequent_flyer_id
                        FROM frequent_flyer

```

```

                                WHERE level =4
                                )
                            )
        SELECT * FROM booking WHERE account_id IN
            (SELECT account_id FROM level4
            )
        )
    SELECT * FROM bk WHERE bk.booking_id IN
        (SELECT booking_id FROM booking_leg WHERE
            leg_num=1 AND is_returning IS false
            AND flight_id IN (
                SELECT flight_id
                FROM flight
                WHERE departure_airport IN ('ORD', 'MDW')
                AND scheduled_departure:: DATE='2023-07-04')
            )
        )
    SELECT count(*) FROM passenger WHERE booking_id IN (
        SELECT booking_id FROM bk_chi)

```

With a query constructed like this, you are not letting the query planner choose the best execution path, because the sequence of actions is hard-coded. Although the preceding query is written in a declarative language, it is imperative by nature.

Instead, to write a declarative query, simply specify what you need to retrieve from the database, as shown in Listing 1-5.

Listing 1-5. Declarative query to calculate the number of passengers

```

SELECT count(*)
FROM booking bk
JOIN booking_leg bl ON bk.booking_id=bl.booking_id
JOIN flight f ON f.flight_id=bl.flight_id
JOIN account a ON a.account_id=bk.account_id
JOIN frequent_flyer ff ON ff.frequent_flyer_id=a.frequent_flyer_id
JOIN passenger ps ON ps.booking_id=bk.booking_id
WHERE level=4
      AND leg_num=1

```



```
AND is_returning IS false
AND departure_airport IN ('ORD', 'MDW')
AND scheduled_departure >= '2023-07-04'
AND scheduled_departure < '2023-07-05'
```

This way, you allow the database to decide which order of operations is best, which may vary depending on the distribution of values in the relevant columns.

You may want to run these queries after all required indexes are built in [Chapter 5](#).

Optimization Goals

So far, we have implied that a performant query is a query that is executed quickly. However, that definition is neither precise nor complete. Even if, for the moment, we consider reduction of execution time as the sole goal of optimization, the question remains: What execution time is “good enough”? For a monthly general ledger of a big corporation, completion within one hour may be an excellent execution time. For a daily marketing analysis, minutes might be great. For an executive dashboard with a dozen reports, refresh within ten seconds may be the best time we can achieve. For a function called from a web application, even a hundred milliseconds can be alarmingly slow.

In addition, for the same query, execution time may vary at different times of day or with different database loads. In some cases, we might be interested in average execution time. If a system has a hard timeout, we may want to measure performance by capping the maximum execution time. There is also a subjective component in response time measurement. Ultimately, a company is interested in user satisfaction. Most of the time, user satisfaction depends on response time, but it is also a subjective characteristic.

However, beyond execution time, other characteristics may be taken into account. For example, a service provider may be interested in maximizing system throughput. A small startup may be interested in minimizing resource utilization without compromising the system’s response time. We know one company that increased the system’s main memory to keep the execution time fast. Their goal was to make sure that the whole database could fit into main memory. That worked for a while until the database grew bigger than any main memory configuration available.

How do we define optimization goals? We use the familiar SMART goal framework. SMART goals are

- Specific
- Measurable
- Achievable (attainable)
- Result-based (relevant)
- Time-bound (time-driven)

Examples of SMART goals are presented in Table 1-1.

Table 1-1. *SMART goal examples*

Characteristic	Bad Example	Good Example
Specific	All pages should respond fast.	Each function execution should be completed before a system-defined timeout.
Measurable	Customers shouldn't wait too long to complete their application.	Response time of the registration page should not exceed four seconds.
Achievable	Daily data refresh time in the data warehouse should never increase.	When source data volume grows, the daily data refresh time should grow not more than logarithmically.
Result-based	Each report refresh should run as fast as possible.	Refresh time for each report should be short enough to avoid lock waits.
Time-bound	We will optimize as many reports as we can.	By the end of the month, all financial reports should run in under 30 seconds.

Optimizing Processes

It is essential to bear in mind that a database does not exist in a vacuum. A database is the foundation for multiple, often independent applications and systems. For any user (external or internal), overall system performance is the one they experience and the one that matters.

At the organization level, the objective is to improve the performance of the whole system. It might be response time or throughput (essential for the service provider) or, most likely, a balance of both. Nobody is interested in database optimizations that have no impact on overall performance.

Database developers and DBAs often tend to over-optimize any bad query that comes to their attention, just because it is bad. At the same time, their work is often isolated from both application development and business analytics. This is one reason optimization efforts may appear to be less productive than they could be. A SQL query cannot be optimized in isolation, outside the context of its purpose and the environment in which it is executed.

Since queries might not be written declaratively, the original purpose of a query might not be evident. Finding out the business intent of what is to be done might be the first and the most critical optimization step. Moreover, questions about the purpose of a report might lead to the conclusion that it is not needed at all. In one case, questioning the purpose of the most long-running reports allowed us to cut the total traffic on the reporting server by 40%.

Optimizing OLTP and OLAP

There are many ways to classify databases, and different database classes may differ in both performance criteria and optimization techniques. Two major classes are *OLTP* (Online Transaction Processing) and *OLAP* (Online Analytical Processing). OLTP databases support applications, and OLAP databases support BI and reporting. Through the course of this book, we will emphasize different approaches to OLTP and OLAP optimization. We will introduce the concepts of *short* queries and *long* queries and explain how to distinguish one from the other.

Hint It does not depend on the length of the SQL statement.

In the majority of cases, in OLTP systems we are optimizing short queries and in OLAP systems both short and long queries.

Database Design and Performance

We have already mentioned that we do not like the approach of “first write, then optimize” and that this book’s goal is to help you write queries right right away. When should a developer start thinking about performance of the query they are working on? The answer is the sooner, the better. Ideally, optimization starts from requirements. In practice, this is not always the case, although gathering requirements is essential.

To be more precise, gathering requirements allows us to come up with the best database design, and database design can impact performance.

If you are a DBA, chances are, from time to time, you get requests to review new tables and views, which means you need to evaluate someone else’s database design. If you do not have any exposure to what a new project is about and the purpose of the new tables and views, there is not much you can do to determine whether the proposed design is optimal. The only thing you may be able to evaluate without going into the details of the business requirements is whether the database design is normalized. Even that might not be obvious without knowing the business specifics.

The only way to evaluate a proposed database design is to ask the right questions. The right questions include questions about what real-life objects the tables represent. To illustrate that statement, let’s look at the following example: in this database, we need to store user accounts, and we need to store each account holder’s phone number(s). Two possible designs are shown in Figures [1-1](#) and [1-2](#), respectively.



Figure 1-1. Single-table design

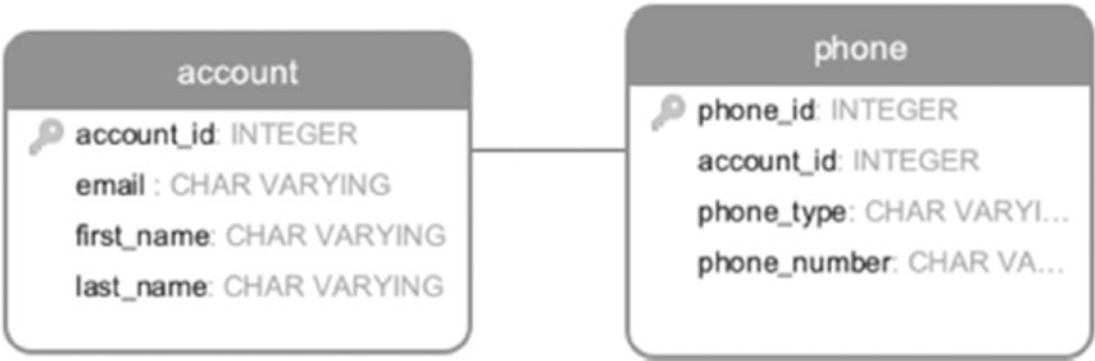


Figure 1-2. Two-table design

Which of the two designs is the right one? It depends on the intended usage of the data. If phone numbers are never used as search criteria and are selected as a part of an account (to be displayed on the customer support screen), if UX has fields labeled with specific phone types, then a single-table design is more appropriate.

However, if we want to search by phone number regardless of type, having all phones in a separate table will make the search more performant.

Also, users are often asked to indicate which phone number is their primary phone. It is easy to add one Boolean attribute `is_primary` to the two-table design, but it will be more complicated in the one-table design. An additional complication might arise when somebody does not have a landline or a work phone, which happens often. On the other hand, people often have more than one cell phone, or they might have a virtual number, like Google Voice, and they might want to record that number as the primary number to reach them. All these considerations are in favor of the two-table design.

Lastly, we can evaluate the frequency of each use case and how critical response time is in each case.

Application Development and Performance

We are talking about application development, not just the database side of development because once again, database queries are not executed by themselves—they are parts of applications. Traditionally, optimizing the individual queries is viewed as “optimization,” but we are going to take a broader approach.

Quite often, although each database query executed by an application returns results in less than 0.1 seconds, an application page response time may amount to ten seconds or more. Technically speaking, optimization of such processes is not a “database optimization” in its traditional meaning, but there is a lot a database developer can do to improve the situation. We cover a relevant optimization technique in Chapters [10](#) and [13](#).

Other Stages of the Lifecycle

The life of an application does not end after release in production, and the optimization is a continuous process as well. Although our goal should be to optimize long-term, it is hard to predict how exactly the system will evolve. It is a good practice to continually keep an eye on the system performance, not only on the execution times but on trends.

A query may be very performant, and one might not notice that the execution time started to increase because it is still within acceptable limits and no automated monitoring system will be alerted.

Query execution time may change because data volume increased or the data distribution changed or execution frequency increased. In addition, we expect new indexes and other improvements in each new PostgreSQL release, and some of them may be so significant that they prompt rewriting original queries.

Whatever the cause of the change is, no part of any system should be assumed to be optimized forever.

PostgreSQL Specifics

Although the principles described in the previous section apply to any relational database, PostgreSQL, like any other database, has some specifics that should be considered. If you have some previous experience in optimizing other databases, you might find a good portion of your knowledge does not apply. Do not consider this a PostgreSQL deficiency; just remember that PostgreSQL does lots of things differently.

For former Oracle users, the most important distinction they should be aware of is that PostgreSQL does not cache queries; it only caches the data in the shared buffers. When Oracle would produce a generic execution plan, which will be available for multiple connections, PostgreSQL will generate a new execution plan each time the query is executed. Even though all the necessary data is already in the main memory, the query itself must be executed. This difference in behavior might drastically decrease application performance after conversion to Oracle. Later in this book, we will discuss the ways to address this problem.

The absence of the cached query plans leads to another important distinction: the difference between the execution of parameterized queries and dynamic SQL. Chapter 13 of this book is dedicated to the use of dynamic SQL, an option that is often overlooked.

Another important PostgreSQL feature you should be aware of is that PostgreSQL does not have optimizer hints. If you previously worked with a database like Oracle, which does have the option of “hinting” to the optimizer, you might feel helpless when you are presented with the challenge of optimizing a PostgreSQL query. However, here is some good news: PostgreSQL does not have hints by design. The PostgreSQL core team believes in investing in developing a query planner that is capable of choosing the best execution path without hints. As a result, the PostgreSQL optimization engine is one of the best among both commercial and open source systems. Many strong database internal developers have been drawn to Postgres because of the optimizer. In addition,

Postgres has been chosen as the founding source code base for several commercial databases partly because of the optimizer. With PostgreSQL, it is even more important to write your SQL statements declaratively, allowing the optimizer to do its job.

With PostgreSQL, it is especially important to be aware of new features and capabilities added with each release. In recent years, Postgres has had over 180 of them each year. Many of these features are around optimization. We are not planning to cover them all; moreover, between the writing of this chapter and its publication, there will indubitably be more. PostgreSQL has an incredibly rich set of types and indexes, and it is always worth consulting recent documentation to check whether a feature you wanted might have been implemented.

More PostgreSQL specifics will be addressed later in the book.

Summary

Writing a database query is different from writing application code using imperative languages. SQL is a declarative language, which means that we specify the desired outcome, but do not specify an execution path. Since two queries yielding the same result may be executed differently, utilizing different resources and taking a different amount of time, optimization and “thinking like a database” are core parts of SQL development.

Instead of optimizing queries that are already written, our goal is to write queries correctly from the start. Ideally, optimization begins at the time of gathering requirements and designing the database. Then, we can proceed with optimizing both individual queries and the way the database calls from the application are structured. But optimization does not end there; in order to keep the system performant, we need to monitor performance throughout the system’s lifecycle and continually seek opportunities to optimize it.

CHAPTER 2

Theory: Yes, We Need It!

In order to write performant queries, database developers need to understand how queries are processed by the database engine. And to do that, we need to know the basics of relational theory. If the word “theory” sounds too dry, we can call it “the secret life of a database query.” In this chapter, we will take a look at this “secret life,” explaining what happens to a database query between the moment you click “Execute” or press Enter and the moment you see the result set returned from the database.

As discussed in the previous chapter, a SQL query specifies what results are needed or what must be changed in the database but does not specify how exactly to execute the requested action. It is the job of the database engine to convert the source SQL query into executable code and execute it. This chapter covers the operations used by the database engine as it interprets a SQL query and their theoretical underpinning.

Query Processing Overview

In order to produce query results, PostgreSQL performs the following steps:

- Compile and transform a SQL statement into an expression consisting of high-level logical operations, known as a logical plan.
- Optimize the logical plan and convert it into an execution plan.
- Execute (interpret) the plan and return results.

Compilation

Compiling a SQL query is similar to compiling code written in an imperative language, such as Java or Python. The source code is parsed, and an internal representation is generated. However, the compilation of SQL statements has two essential differences.

First, in an imperative language, the definitions of identifiers are usually included in the source code, while definitions of objects referenced in SQL queries are mostly stored in the database. Consequently, the meaning of a query depends on the database structure: for example, an object with the same name could be a table in one database, a view in another one, and a function in yet another one.

Second, the output of an imperative language compiler is usually (nearly) executable code, such as byte code for a Java virtual machine. In contrast, the output of a query compiler is an expression consisting of high-level operations that remain declarative—they do not give any instruction on how to obtain the required output. A possible order of operations is specified at this point, but not the manner of executing those operations.

Optimization and Execution

The instructions on how to execute the query appear at the next phase of query processing, optimization. The optimizer performs two kinds of transformations: it replaces logical operations with their execution algorithms and possibly changes the logical expression structure by changing the order in which logical operations will be executed.

Neither of these transformations is straightforward; a logical operation can be computed using different algorithms, and the optimizer tries to choose the best one. The same query may be represented with several equivalent expressions producing the same result but requiring a significantly different amount of computational resources for execution. The optimizer tries to find a logical plan and physical operations that minimize required resources, including execution time. This search requires sophisticated algorithms that are out of scope for this book. However, we do cover how an optimizer estimates the amount of resources needed for physical operations and how these resources depend on the specifics of how data is stored and the amount of data stored.

The output of the optimizer is an expression containing physical operations. This expression is called a (physical) execution plan. For that reason, the PostgreSQL optimizer is called the query planner.

Finally, the query execution plan is interpreted by the query execution engine, frequently referred to as the executor in the PostgreSQL community, and output is returned to the client application.

Let's take a closer look at each step of query processing and the operations each uses.

Relational, Logical, and Physical Operations

To go deeper into how SQL is understood by the database engine, we must at last confront this chapter's titular concern: theory. Many modern database management systems, including PostgreSQL, are called relational because they are based on relational theory.¹ Despite some bad press (that theory is dry, incomprehensible, or irrelevant), understanding a small part of relational theory is essential to master optimization—specifically, relational operations. To be more precise, we will need to understand how relational operations correspond to logical operations and the query language used in queries. The previous section covered three steps of query processing at a high level; this section describes each level in more detail, starting with descriptions of relational operations.

Some readers may think the material covered here is trivial and find it already familiar, while others may feel that this is introducing an unnecessary complication. For now, hang in there and trust that this is building a foundation for what comes next.

Relational Operations

The central concept of relational theory is a *relation*. For our purposes, we view a relation as a table, although academics may quibble that this elides some subtle but important differences.

Any relational operation takes one or more relations as its arguments and produces another relation as its output. This output can be used as an argument for another relational operation producing yet another relation that, in turn, can become an argument. This way, we can build complex expressions and represent complex queries. The possibility to construct complex expressions makes the set of relational operations (called *relational algebra*) a powerful query language.

Moreover, expressions in relational algebra can be used to define additional operations.

¹C. J. Date, *An Introduction to Database Systems*; J. Ullman, *Principles of Database Systems*, Second Edition

The first three operations to be discussed are *filter*, *project*, and *product*.



Figure 2-1. *Filter*

The *filter* operation (represented in Figure 2-1) is often called selection and is called restriction in relational theory. We prefer to use the term *filter* to avoid confusion with the SQL SELECT statement, while the term *restriction* has too deep of mathematical origins. This operation accepts a single relation as an argument and includes in its output all tuples (or rows) satisfying the condition specified as a filtering condition, for example:

```
SELECT *  
FROM flight  
WHERE departure_airport='LAG'  
      AND (arrival_airport='ORD'  
          OR arrival_airport='MDW')  
      AND scheduled_departure BETWEEN '2023-05-27' AND  
                                     '2023-05-28'
```

Here, we start from the relation `flight` and apply restrictions on the values of `arrival_airport`, `departure_airport`, and `scheduled_departure` attributes. The result is a set of records, that is, also a relation.

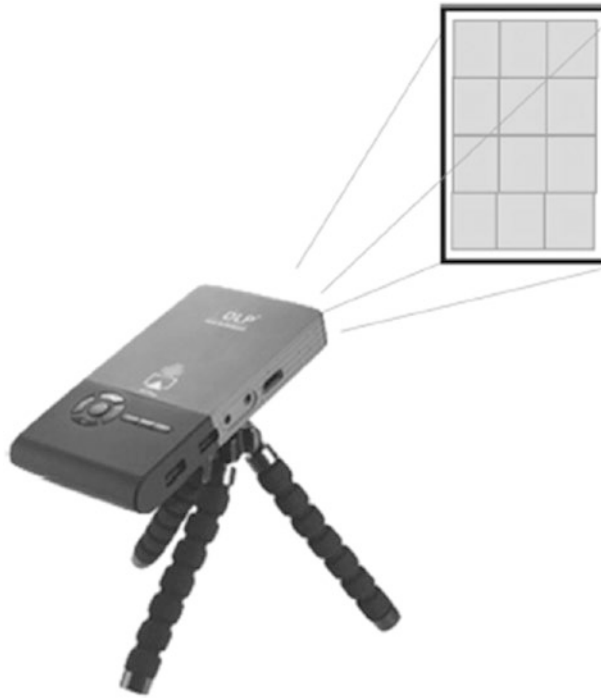


Figure 2-2. *Project*

The *project* operation (represented in Figure 2-2) similarly takes a single relation as an argument and removes some attributes (columns). The relational project operation also removes duplicates from the output, while the SQL project operation does not, for example:

```
SELECT city, zip FROM address
```

when executed in PostgreSQL will return as many rows as there are records in the address table. But if we perform the relational operation project, it would leave one record for each zip code. To achieve the same result in PostgreSQL, we would need to add the *distinct* keyword:

```
SELECT DISTINCT city, zip FROM address
```

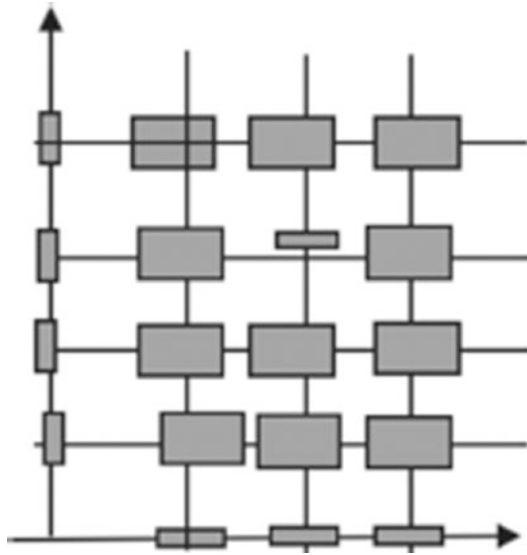


Figure 2-3. *Product*

The *product* operation (also called a Cartesian product and represented by Figure 2-3) produces the set of all pairs of rows from its first and second arguments. It is very difficult to find a real-life, useful example of a product, but let's imagine we want to find all possible flights that might exist (from any airport in the world to any airport in the world). The product operation will look like this:

```
SELECT
    d.airport_code AS departure_airport
    a.airport_code AS arrival_airport
FROM  airport a,
      airport d
```

Now that we have covered these primary relational operations, you may feel cheated: Where is the join operation? We know that join operations are essential. The answer is hidden in plain sight: a join operation can be expressed as a product followed by filtering. From a relational theory point of view, a join operation is redundant. This is a perfect example of how a declarative language works; the formal definition is one way (but not the only way) to find the result of a join. If we compute a Cartesian product of two relations and then apply a filter, we will obtain the desired result. But hopefully,

no database engine would use this approach on a larger dataset; it could literally take years! In Chapter 3, we will discuss how joins can be implemented more efficiently than straightforward computation based on the formal definition.

Relational operations also include grouping, union, intersection, and set difference.

The last piece of relational theory that we need for optimization is *equivalence rules*. All relational operations satisfy several equivalence rules, including

- *Commutativity* – $\text{JOIN}(R, S) = \text{JOIN}(S, R)$

Commutativity means that the order of two relations is not important. If we have two relations, R and S, then R JOIN S will produce the same result as S JOIN R.

- *Associativity* – $\text{JOIN}(R, \text{JOIN}(S, T)) = \text{JOIN}(\text{JOIN}(R, S), T)$

Associativity means that if we have three relations, R, S, and T, we can choose to first perform R JOIN S and then JOIN T to the result, or we can first perform S JOIN T and then JOIN R to the result of the first JOIN, and the results will be equivalent in both cases.

- *Distributivity* – $\text{JOIN}(R, \text{UNION}(S, T)) = \text{UNION}(\text{JOIN}(R, S), \text{JOIN}(R, T))$

Distributivity means that if we are joining a relation with a UNION of two other relations, the result will be the same as when we perform two joins, R JOIN S and R JOIN T separately, and then UNION the results.

The equivalence rules listed in the preceding text are just examples among dozens. Why is it important to know about these rules? For efficiency, it might be better to execute operations in a different order than they are listed in. There will be multiple examples of such transformations in subsequent chapters. Equivalences ensure that a query may be represented with several different expressions, providing the impetus for an optimizer.

Logical Operations

The set of logical operations needed for representation of SQL queries includes all relational operations, but the semantics is different. As noted earlier, the SQL project operation does not remove duplicates. An additional operation for removal of duplicates is included.

Other additional operations are needed to represent SQL constructs that cannot be expressed in relational theory, for example, left, right, and full outer joins produce a result that is not a relation (but still is a SQL table).

Many equivalence rules are also valid for logical operations. For any relatively complex query, an optimizer can choose the best from a huge number of expressions. More information about relational theory can be found in the resources found in the end notes.

Queries as Expressions: Thinking in Sets

Writing declarative queries is not an easy task for humans. People are more familiar with actions than with rules or conditions. Thinking in sets² makes it easier: we can think about actions on tables and operations on tables, rather than on individual objects (or rows).

All logical operations mentioned earlier can be easily expressed in SQL. These operations accept tables as arguments, both tables stored in the database and tables that are the result output of previous operations.

A PostgreSQL expression written as a SQL query will be processed by the optimizer and will mostly likely be replaced with another, equivalent expression, using the equivalence rules discussed earlier.

Since the result of any relational operation is a relation, it can be passed directly to the next relational operation, without the need for intermediate storage. Some database developers choose to create temporary tables to store intermediate results, but such practices can produce unneeded computational overhead and block the optimizer.

In more theoretical words, the previous paragraph states that the ability of an optimizer to produce an efficient execution plan depends on two factors:

- A rich set of equivalences provides for a large space of equivalent expressions.
- Relational operations produce no side effects, such as temporary tables—that is, the only thing produced is the result of the operation.

²Joe Celko, *Joe Celko's Thinking in Sets: Auxiliary, Temporal, and Virtual Tables in SQL* (The Morgan Kaufmann Series in Data Management Systems)

Operations and Algorithms

In order to make a query executable, logical operations must be replaced with physical operations (also called algorithms). In PostgreSQL, this replacement is performed by the query planner, and the overall execution time of a query depends on which algorithms are chosen and whether they are chosen correctly.

When we move from the logical to the physical level, mathematical relations are transformed into tables that are stored in the database, and we need to identify ways to retrieve data from the tables. Any stored data must be extracted with one of the *data access algorithms* discussed in the next chapter. Usually, data access algorithms are combined with operations consuming their results.

More complex logical operations, such as join, union, and grouping, can be implemented with several alternative algorithms. Sometimes a complex logical operation is replaced with multiple physical operations.

These algorithms are discussed in detail in [Chapter 3](#).

Summary

The database engine interprets SQL queries by parsing them into a logical plan, transforming the results, choosing algorithms to implement the logical plan, and finally executing the chosen algorithms. The logical operations used by the database engine are based on operations derived from relational theory, and understanding these is crucial to thinking like a database.

CHAPTER 3

Even More Theory: Algorithms

By now, those of you who are diligently reading this book without skipping chapters might be impatient. We are already in Chapter 3, and we are still talking about theory! When are we going to get to write code?

Very soon! This chapter covers the last part of query processing, and by the end, we will have all the pieces we need to understand execution plans.

Chapter 2 covered relational operations and stated that we need physical operations, or algorithms, to execute queries. Mapping these algorithms to logical operations is not straightforward; sometimes, a complex logical operation is replaced with multiple physical operations, or several logical operations are merged into a single physical operation.

This chapter describes those algorithms, starting from algorithms for data retrieval and then proceeding to algorithms for more complex operations.

Understanding these algorithms will allow us to go back to execution plans and get a better grasp of their components. Thus, we will be only one step away from our goal: learning how to tune queries.

Algorithm Cost Models

Chapter 1 mentioned several ways of measuring the performance of a system, including response time, cost, and user satisfaction. These metrics are external to the database, and although external metrics are the most valuable, they aren't available to the query optimizer.

Instead, the optimizer uses internal metrics based on the amount of computing resources needed to execute a query or a single physical operation within a plan. The most important resources are those that affect execution time, namely, CPU cycles and I/O accesses (read/write disk blocks). Other resources, such as memory or disk space, have an indirect impact on execution time; for example, the amount of available memory will influence the ratio of CPU cycles and I/O accesses. The distribution of memory is almost always controlled by configuration parameters and will be covered in Chapter 10.

These two primary metrics, CPU cycles and number of I/O operations, are not directly comparable. However, in order to compare query execution plans, the optimizer has to combine them into a single cost function: the lower the cost, the better the plan. For several decades, the number of I/O operations was the dominating component of the cost because rotating hard drives are orders of magnitude slower than CPUs. This is not necessarily the case for modern hardware, so the optimizer must be tuned to use the correct ratio. This is also controlled via server parameters.

A cost model of a physical operation estimates the resources needed to execute the operation. In this book we outline simplified cost models that are easy to understand but are still useful in understanding how more sophisticated cost models work. Specifically, we'll estimate the amount of resources needed for an algorithm as the number of low-level operations to be performed for each row or each block of tables used as an input or produced as an output of an algorithm. For example, we count the number of block accesses but do not distinguish if the access actually requires an I/O operation. In our simplified models, an access to a block is an access, whether it requires I/O or the block is already on the buffer pool of the operating system.

Generally, the cost depends on the tables given as arguments to the operation. To represent cost models, we'll use simple formulas with the following notation: for any table or relation R , TR and BR denote the number of rows in the table and the number of storage blocks occupied by the table, respectively. Additional notation will be introduced as needed.

The following section discusses physical operations, outlining algorithms and cost models for each. As the relative speed of CPU and external storage may vary in a wide range, CPU costs and I/O costs are considered separately. Two logical operations discussed in the previous chapter, project and filter, are not included. These are typically combined with the operation that precedes them, because they can be applied independently to a single row, without depending on other rows in the argument table.

Of course, the project and filter are not free in terms of cost that is proportional to the number of processed rows. However, in most cases these costs are negligible, so we do not include these costs into cost models of other algorithms.

Data Access Algorithms

To begin executing a query, the database engine must extract stored data. This section concerns algorithms used to read data from database objects. In practice, these operations are often combined with their following operation in the query execution plan. This is advantageous in cases where it is possible to save execution time by avoiding reading that will be subsequently filtered out.

The efficiency of such operations depends on the ratio of rows that are retained to the total rows in the stored table. This ratio is called *selectivity*. The choice of algorithm for a given read operation depends on the selectivity of filters that can be simultaneously applied.

Storage Structures

It should come as no surprise that data is stored in files that reside on hard drives. Any file used for database objects is divided in blocks of the same length; by default, PostgreSQL uses blocks containing 8192 bytes each. A block is the unit that is transferred between the hard drive and the main memory, and the number of I/O operations needed to execute any data access is equal to the number of blocks that are being read or written.

Database objects consist of logical items (table rows, index records, etc.). PostgreSQL allocates space for these items in blocks. Several small items can reside in the same block; items that don't fit within a single block use a separate storage mechanism. The generic structure of a block is shown in Figure 3-1.

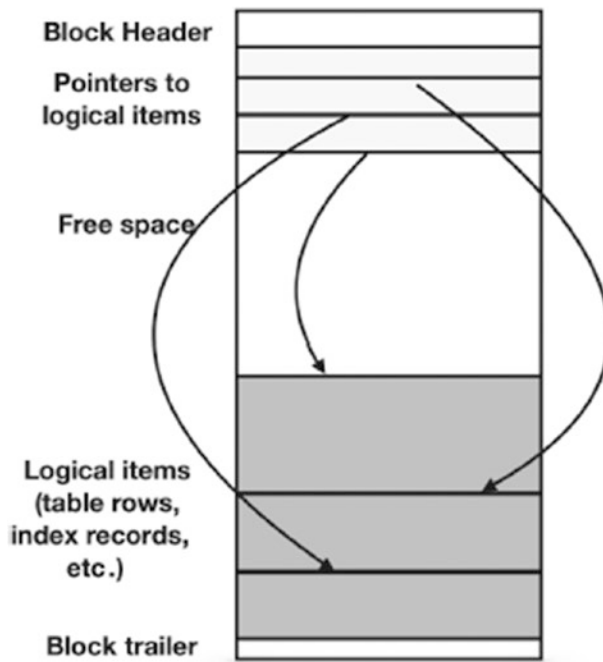


Figure 3-1. *The generic block structure in PostgreSQL*

The allocation of items to blocks also depends on the type of the database object. Table rows are stored using a data structure called a heap: a row can be inserted in any block that has sufficient free space, without any specific ordering. Other objects (e.g., indexes) may use blocks differently.

Full Scan

In a full scan, at times referred to as a sequential scan, the database engine consecutively reads all of the rows in a table and checks the filtering condition for each row. To estimate the cost of this algorithm, we need a more detailed description, as shown in the pseudocode in Listing 3-1.

Listing 3-1. Pseudocode for a full-scan data access algorithm

```
FOR each block IN a_table LOOP
    read block;
    FOR each row IN block LOOP
        IF filter_condition (row)
```

```

        THEN output (row)
      END IF;
    END LOOP;
  END LOOP;

```

The number of I/O accesses is BR; the total number of iterations of the inner loop is TR. We also need to estimate the cost of operations producing the output. This cost depends on selectivity, denoted as S , and is equal to $S * TR$. Putting all these parts together, we can estimate the cost of a full scan as

$$c1 * BR + c2 * TR + c3 * S * TR$$

where constants $c1$, $c2$, and $c3$ represent properties of hardware.

A full scan can be used with any table; additional data structures are not needed. Other algorithms depend on the existence of indexes on the table, described in the following.

Index-Based Table Access

Note that until we got to physical operations, we did not even mention data access algorithms. We do not need to “read” relations—they are abstract objects. If we follow the idea that relations are mapped to tables, there is no other way to retrieve data than to read the whole table into the main memory. How else will we know which rows of data contain which values? But relational databases wouldn’t be such a powerful tool for data processing if we stopped there. All relational databases, including PostgreSQL, allow for building additional, redundant data structures, making data access dramatically faster than a simple sequential read.

These additional structures are called indexes.

How indexes are built will be covered later in this chapter; for now, we need to understand two facts about indexes. First, they are “redundant” database objects; they do not store any additional information that can’t be found in the source table itself.

Second, indexes provide additional data access paths; they allow us to determine what values are stored in the rows of a table without actually reading the table—this is how index-based access works. And, as mentioned previously, this happens entirely invisibly to the application.

If a filtering condition (or conditions) is encapsulated by an index on a table, the index can be used to access data from that table.

To get a table row from a pointer, the block containing this row must be read. The underlying data structure of a table is a *heap*, that is, rows are stored unordered. Their order is not guaranteed, nor does it correspond to properties of the data. Depending on the index selectivity, PostgreSQL utilizes one of two methods. The first is an `index scan`; the second depends on a `bitmap index scan`, followed by a `bitmap heap scan`.

An index entry includes addresses of the records that match the indexed value; this address contains the block address and the offset of the record within the block. In an `index scan`, the database engine reads each entry of the index that satisfies the filter condition and retrieves blocks in index order. Because the underlying table is a heap, multiple index entries might point to the same block. If an index has a low selectivity, these situations are rare, and there are high chances that no block will be read (or rather attempted to read) more than once. However, the situation is different when the selectivity is high or multiple indexes are used or the filtering condition differs from equal. In these cases, a `bitmap index scan` will be used. This method builds a heap bitmap in memory. The whole bitmap is a single bit array, with as many bits as there are heap blocks in the table being scanned.

Each index used to satisfy a selection criterion is used sequentially. To perform a `bitmap index scan`, first, a bitmap is created with all entries initially set to 0 (false). Whenever an index entry that matches the search condition is found, the bit corresponding to the heap block indicated by the index entry is set to 1 (true). The second and any additional `bitmap index scans` do the same thing with the indexes corresponding to the additional search conditions. Once all bitmaps have been created, the engine performs a bitwise logical AND operation to find which blocks contain requested values for all selection criteria, producing a final candidate list. This means that blocks that satisfy only one of the two criteria in a logical AND never have to be accessed. An illustration is shown in Figure 3-2.

Block#	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Index 1	0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	1
Index 2	0	0	1	0	0	1	0	0	0	1	0	0	1	0	0	1
Logical AND ↓																
	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1

Figure 3-2. Using bitmaps for table access through multiple indexes

After the final candidate list is computed, the candidate blocks are read sequentially using a `bitmap heap scan` (a heap scan based on a bitmap), and for each block, the individual records are examined to recheck the search conditions. Note that requested values may reside in different rows in the same block. The bitmap ensures that relevant rows will not be missed, but does not guarantee that all scanned blocks contain a relevant row. This explanation elides some implementation details for the sake of brevity and clarity. The actual implementation is more sophisticated and handles various edge cases.

The cost model of index-based access is much more complex than a full scan. Informally, it can be described this way: For small values of selectivity, rows satisfying the filtering conditions will most likely be located in different blocks and, consequently, the cost is proportional to the number of result rows. For larger values of selectivity, the number of processed blocks approaches the total number of blocks. In the latter case, the cost becomes higher than the cost of a full scan because resources are needed to access the index.

Index-Only Scan

Data access operations do not necessarily return entire rows. If some columns are not needed for the query, these columns can be skipped as soon as a row passes filtering conditions (if any). More formally, this means that the logical project operation is combined with data access. This combination is especially useful if an index used for filtering contains every column needed for the query.

In this algorithm, data is read from the index and any remaining filtering conditions are applied, if necessary. Usually there is no need to access table data, but sometimes additional checks are needed—this will be discussed in detail in [Chapter 5](#).

The cost model for an index-only scan is similar to the model for index-based table access except that there's no need to actually access table data. For small values of selectivity, the cost is approximately proportional to the number of returned rows. For large values of selectivity, the algorithm performs an (almost) full scan of the index. The cost of an index scan is usually lower than the cost of a full table scan because it contains less data.

Comparing Data Access Algorithms

The choice of the best data access algorithm depends mostly on query selectivity. The relationship of cost to selectivity for different data access algorithms is shown in Figure 3-3. We intentionally omitted all numbers on this chart as they depend on hardware and table size, while the qualitative comparison does not.

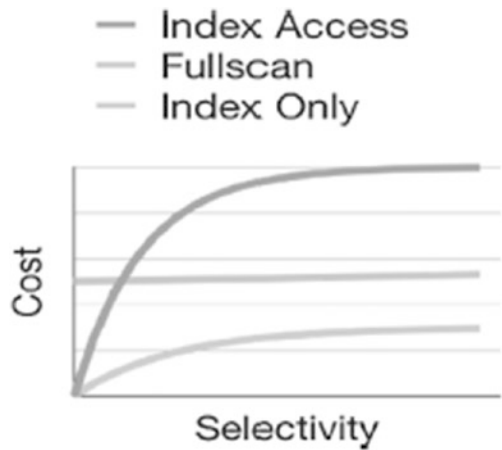


Figure 3-3. Relationship of cost and query selectivity for different data access algorithms

The cost function for a full scan is linear and is almost horizontal because the growth is due to generation of output. Typically, the cost of output generation is negligible in comparison with other costs for this algorithm.

The cost function of index-based table access starts from (almost) 0 and grows quickly with the growth of selectivity. The growth slows down for large values of selectivity, where the cost is significantly higher than the cost of a full scan.

The most interesting point is the intersection of the two functions: for smaller values of selectivity, index-based access is preferable, while a full scan is better for larger values of selectivity. The position of the intersection depends on hardware and may depend on the size of the table. For relatively slow rotating drives, index-based access is preferable only if selectivity does not exceed 2–5%. For SSDs or virtual environments, this value can be higher. On older spinning disk drives, random block access can be an order of magnitude slower than sequential access, so the additional overhead of indexes is higher for a given proportion of rows.

The line representing an index-only scan is the lowest, meaning that this algorithm is always preferable if it is applicable (i.e., all needed columns are in the index).

The query optimizer estimates both the selectivity of a query and the selectivity of the intersection point for this table and this index. The query shown in Listing 3-2 has a range filtering condition that selects a significant portion of the table.

Listing 3-2. A range filtering query executed with a full table scan

```
SELECT
    flight_no,
    departure_airport,
    arrival_airport
FROM flight
WHERE scheduled_departure BETWEEN
'2023-05-15' AND '2023-08-31';
```

In this case, the optimizer chooses a full scan.

However, a smaller range in the same query (see Listing 3-3) results in index-based table access.

Listing 3-3. Range filtering with index-based table access

```
SELECT
    flight_no,
    departure_airport,
    arrival_airport
FROM flight
WHERE scheduled_departure BETWEEN
'2023-08-12' AND '2023-08-13';
```

In reality, the job of a query optimizer is much more complex: filtering conditions can be supported with multiple indexes with different values of selectivity. Multiple indexes can be combined to produce a block bitmap with fewer number of blocks to be scanned. As a result, the number of choices available to the optimizer is significantly larger than three algorithms.

Thus, there are no winners and losers among data access algorithms. Any algorithm can be a winner under certain conditions. Further, the choice of algorithm depends on storage structures and statistical properties of the data. The database maintains

metadata known as statistics for tables including metrics such as column cardinality, sparseness, etc. Usually these statistics are not known during application development and may change throughout the application lifecycle. Therefore, the declarative nature of the query language is essential for system performance. More specifically, as the table statistics change or if other costing factors are adjusted, a different execution plan can be chosen for the same query.

Index Structures

This section briefly explores the most common index structures, such as trees and hash indexes, and touches on some PostgreSQL specifics.

We show how to estimate the scale of improvement for different types of indexes and how to detect cases when index usage won't provide any performance benefits.

What Is an Index?

One might assume that any person who works with databases knows what an index is. Alas, a surprising number of people, including database developers and report writers and, in some cases, even DBAs, use indexes, even create indexes, with only a superficial understanding of what indexes are and how they are structured. To avoid misunderstanding, we'll begin with a definition of what we mean by an index.

There are many types of indexes, so it is foolhardy to search for structural properties to recognize an index. Instead, we define an index based on its usage. A data structure is called an index if it is

- A redundant data structure
 - Invisible to the application
 - Designed to speed up data selection based on certain criteria
-

The redundancy means that an index can be dropped without any data loss and can be reconstructed from data stored elsewhere (in the tables, of course). Invisibility means that an application cannot detect if an index is present or absent. That is, any query

produces the same results with or without an index. And finally, an index is created with the hope (or confidence) that it improves performance of a specific query or (even better!) several queries.

The performance improvement does not come for free. As an index is redundant, it must be updated when table data are updated. That produces some overhead for update operations that is sometimes not negligible. In particular, PostgreSQL indexes may have an outsized impact on vacuum operations. However, many database textbooks overestimate this overhead. Modern high-performance DBMSs use algorithms that reduce the cost of index updates, so usually, it is beneficial to create several indexes on a table.

Although index structures can differ significantly among index types, the speed-up is achieved due to a fast check of some filtering conditions specified in a query. Such filtering conditions specify certain restrictions on table attributes.

Figure 3-4 shows how an index can speed up the access to the specific table rows.

The right part of Figure 3-4 shows a table, and the left represents an index that can be viewed as a special kind of a table. Each row of the index consists of an index key and a pointer to a table row. The value of an index key usually is equal to the value of a table attribute. The example in Figure 3-4 has airport code as its value; hence, this index supports search by airport code.

A column can have the same value in multiple rows of a table. If this column is indexed, the index must contain pointers to all rows containing this value of an index key. That is, a key may be logically associated with a list of pointers to rows rather than a single pointer.

In PostgreSQL, for some of the index types, an index contains multiple records, that is, the index key is repeated for every pointer to a table row.

Figure 3-4 explains how to reach the corresponding table row when an index record is located; however, it does not explain why an index row can be found much faster than a table row. Indeed, this depends on how the index is structured, and this is exactly what is discussed in the following subsections.

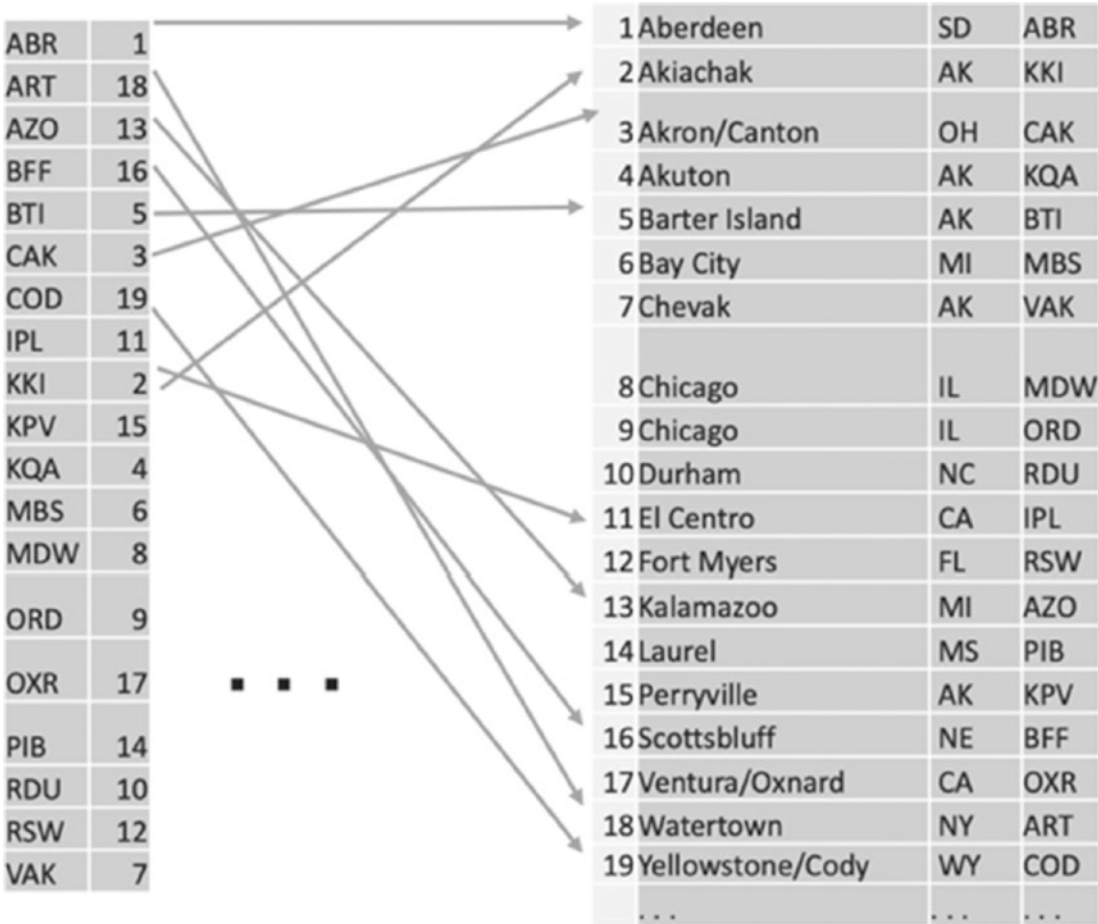


Figure 3-4. Index access to table rows

B-Tree Indexes

The most common structure of an index is a B-tree. The structure of a B-tree is shown in Figure 3-5; airport codes are the index keys. The tree consists of hierarchically organized nodes that are associated with blocks stored on a disk.

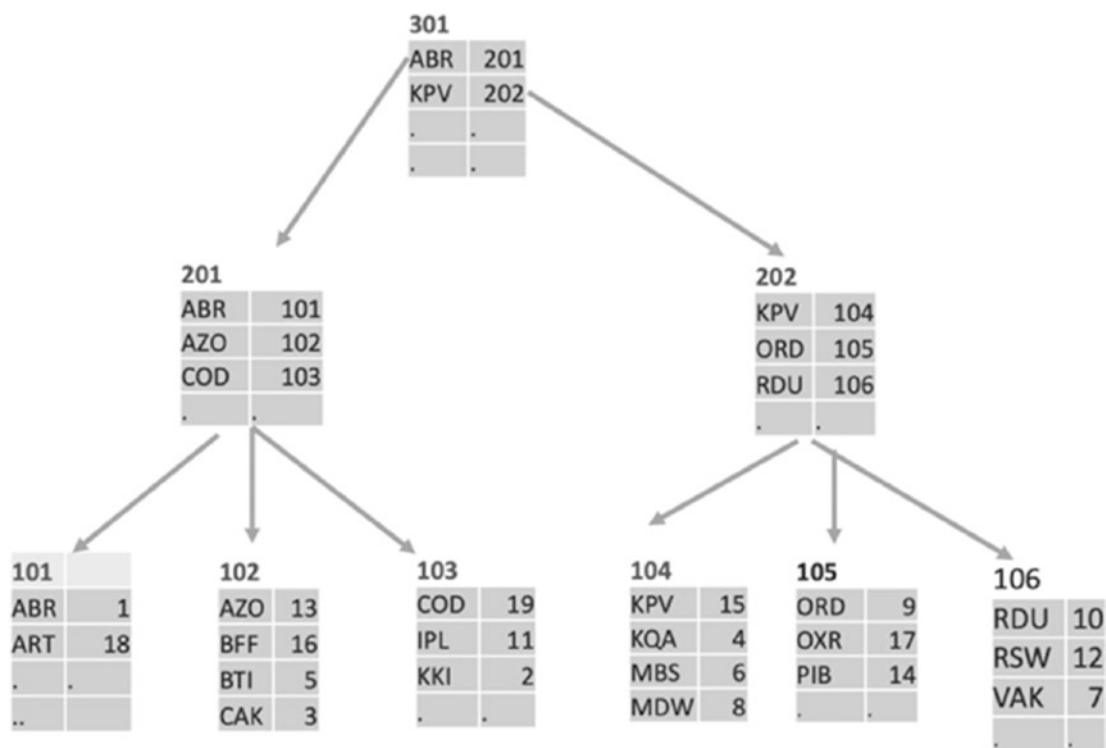


Figure 3-5. An example of a B-tree

The leaf nodes (shown in the bottom row in Figure 3-5) contain index records exactly like those in Figure 3-4; these records contain an index key and a pointer to a table row. Non-leaf nodes (located at all levels except the bottom) contain records that consist of the smallest key (in Figure 3-5, the lowest alphanumeric value) in a block located at the next level and a pointer to this block. All records in all blocks are ordered, and at least half of the block capacity is used in every block.

Any search for a key K starts from the root node of the B-tree. During the block lookup, the largest key P not exceeding K is found, and then the search continues in the block pointed to by the pointer associated with P until the leaf node is reached, where a pointer refers to table rows. The number of accessed nodes is equal to the depth of the tree. Of course, the key K is not necessarily stored in the index, but the search finds either the key or the position where it could be located.

B-trees also support range search (expressed as a between or inequality operations in SQL). As soon as the lower end of the range is located, all index keys in the range are obtained with a sequential scan of leaf nodes until the upper end of the range is reached. A scan of leaf nodes is also needed to obtain all pointers if the index is not unique (i.e., an index value may correspond to more than one row).

Why Are B-Trees Used So Often?

We know from computer science that no lookup algorithm can find an index key among N different keys faster than in $\log N$ time (measured in CPU instructions). This performance is achieved with binary search on an ordered list or with binary trees. However, the cost of updates (such as insertions of new keys) can be very high for both ordered lists and binary trees: an insertion of a single record can cause complete restructuring. This makes both structures unusable for external storage.

In contrast, B-trees can be modified without significant overhead. When a record is inserted, the restructuring is limited to one block. If the block capacity is exceeded, then the block is split into two blocks, and the update is propagated to upper levels.

In the worst case, the number of modified blocks cannot exceed the depth of the tree.

To estimate the cost of a B-tree search, we need to calculate the depth. If each block contains f pointers, then the number of blocks at each level is f times larger than in the previous one. Consequently, the depth of a tree containing N records is $\log N / \log f$. This formula gives the number of block accesses needed for a single key search. The number of CPU instructions is limited for each block, and usually binary search is used inside a block.

Consequently, the CPU cost is only slightly worse than the best that is theoretically possible. Block size in PostgreSQL is 8 Kb. An 8 Kb block can fit dozens of index records; consequently, an index with six to seven levels can accommodate billions of index records.

The depth of the tree is critical for performance of the index. After a split of an overfull block, each of new blocks is at least half-filled. As a result, the depth of the tree does not grow significantly, and the index remains efficient. Theoretically, blocks that are not sufficiently filled should be merged. However, none of actual implementations (including PostgreSQL) do the merge. Consequently, a large number of deletions may result in too large depth and degrade the performance. To avoid the degradation, the index should be rebuilt.

In PostgreSQL, a B-tree index can be created for any ordinal data type; that is, for any two distinct values of the data type, one value is less than the other. This includes user-defined types.

Other Kinds of Indexes

PostgreSQL offers a variety of index structures supporting several data types and several classes of search conditions.

A *hash index* uses a hash function to calculate the address of an index block containing an index key. This type of index has better performance than a B-tree index for equality conditions. However, this index is completely useless for range queries. The cost estimation for hash index search does not depend on index size (in contrast with logarithmic dependency for B-trees).

An *R-tree index* supports a search on spatial data. An index key for an R-tree always represents a rectangle in a multidimensional space. A search returns all objects having a non-empty intersection with the query rectangle. The structure of an R-tree is similar to the structure of a B-tree; however, splitting overflowed nodes is much more complicated. R-tree indexes are efficient for a small number of dimensions (typically, two to three).

Other types of indexes available in PostgreSQL are useful for full text search, search in very large tables, and much more. Additional details on these topics are covered in Chapter 14. Any of these indexes can be relatively easily configured for user-defined data types. However, we do not discuss indexes on user-defined types in this book.

Combining Relations

The real power of relational theory and SQL databases relies on combining data from several tables.

In this section, we describe algorithms for operations that combine data, including a Cartesian product, joins, union, intersection, and even grouping. Surprisingly, most of these operations can be implemented with almost identical algorithms. For this reason, we discuss algorithms rather than the operations they implement. We will use the names *R* and *S* for input tables when describing these algorithms.

Nested Loops

The first algorithm is for a Cartesian product, that is, the set of all pairs of rows from the input tables. The easy way to calculate the product is to loop over table R and, for each row of R, loop over S. The pseudocode for this simple algorithm is presented in Listing 3-4, and the graphical representation of the algorithm is shown in Figure 3-6.

Listing 3-4. Pseudocode for nested loops

```
FOR row1 IN table1 LOOP
  FOR row2 IN table2 LOOP
    INSERT output row
  END LOOP
END LOOP
```

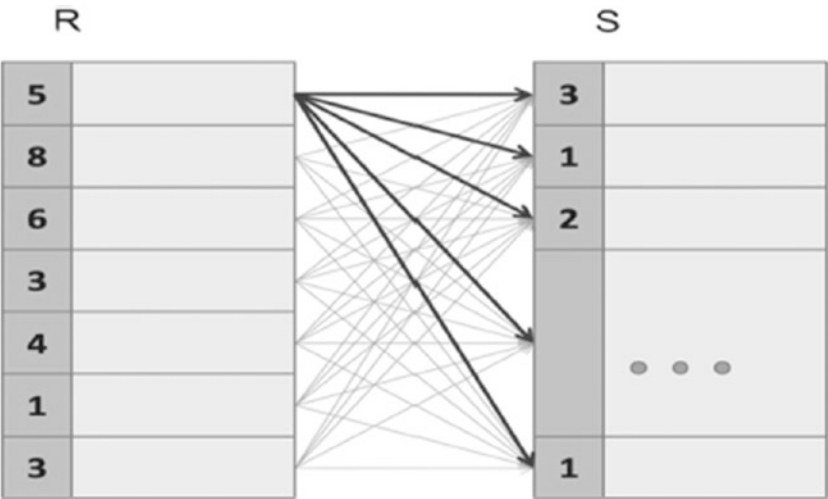


Figure 3-6. Nested loop algorithm

The time needed for this simple algorithm is proportional to the product of the sizes of the input tables: $\text{rows}(R) * \text{rows}(S)$.

A remarkable theoretical fact states that any algorithm that calculates a Cartesian product cannot perform better; that is, any algorithm’s cost will be proportional to the product of the sizes of its inputs or higher. Of course, some variations of this algorithm may perform better than others, but the cost remains proportional to the product.

Slight modifications of the nested loop algorithm can calculate nearly any logical operation that combines data from two tables. The pseudocode in Listing 3-5 implements the join operation.

Listing 3-5. Nested loop algorithm for a join operation

```
FOR row1 IN table1 LOOP
    FOR row2 IN table2 LOOP
        IF match(row1,row2) THEN
            INSERT output row
        END IF
    END LOOP
END LOOP
```

Observe that a nested loop join is a straightforward implementation of the abstract definition of a join, as a Cartesian product followed by a filter. As the nested loop join processes all pairs of rows from the input, the cost remains the same, although the size of the output is smaller than in the case of a Cartesian product.

In practice, one or both input tables are stored tables, rather than the result of preceding operations. If this is the case, a join algorithm can be combined with data access.

Although the processing cost remains the same, variations of the nested loop algorithm combined with a full scan execute nested loops on blocks of input tables and another level of nested loops on rows contained in these blocks. More sophisticated algorithms minimize the number of disk accesses by loading multiple blocks of the first table (outer loop) and processing all rows of these blocks with a single pass over *S*.

These algorithms can work with any join conditions. However, the majority of joins we will ever need to execute are equi-joins, that is, the join condition requires that some attributes of *R* are equal to the corresponding attributes of *S*.

The nested loop join algorithm can also be combined with index-based data access if the table *S* has an index on attributes used in the join condition. For equi-joins, the inner loop of the index-based nested loop algorithm shrinks to few rows of *S* for each row of *R*. The inner loop can even vanish completely if the index on *S* is unique, for example, the join attribute of *S* is its primary key.

The index-based nested loop algorithm is usually the best choice if the number of rows in *R* is also small. However, index-based access becomes inefficient if the number of rows to be processed becomes high, as discussed in Chapter 2.

It is possible to formally prove that there does not exist an algorithm more performant for Cartesian products and joins with arbitrary conditions than nested loops. However, the important question is whether there exists a better algorithm for any specific types of join conditions. The next section shows this is true for equi-joins.

Hash-Based Algorithms

The output of an equi-join consists of pairs of rows from R and S that have equal values on the join attributes. The idea of the hash join algorithm is simple: if the values are equal, then the hash values are also equal.

The algorithm partitions both input tables according to values of the hash function and then joins rows in each bucket independently. The schema of this algorithm is shown in Figure 3-7.

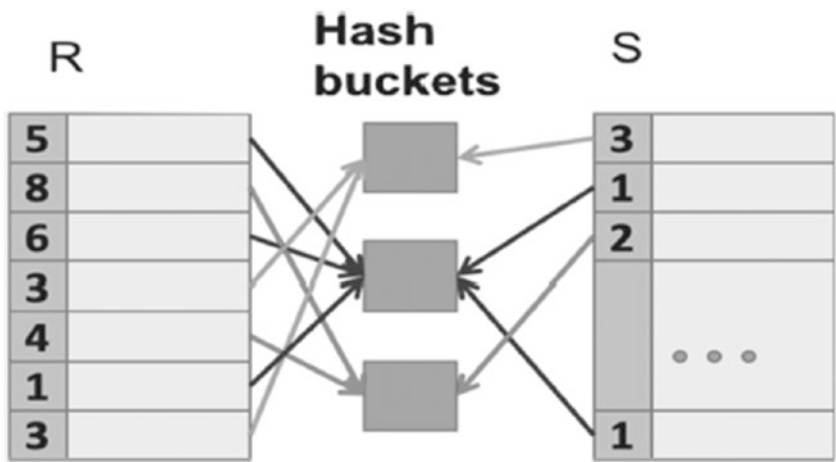


Figure 3-7. Hash join algorithm

The basic version of the hash join algorithm includes two phases:

1. During the *build* phase, all tuples of R are stored in buckets according to the values of the hash function.
2. In the *probe* phase, each row of table S is sent to an appropriate bucket. If matching rows of table R are in the bucket, output rows are produced.

The easiest way to find matching rows in the bucket is to use nested loops (actually loop over all rows in the bucket for each row of S).

The two phases of the hash-based algorithm are shown as separate physical operations in the execution plan.

The cost of a hash join can be approximately estimated with the following formula, where JA is the join attribute:

$$\text{cost}(\text{hash}, R, S) = \text{size}(R) + \text{size}(S) + \text{size}(R) * \text{size}(S) / \text{size}(\text{JA})$$

The first and second terms in this formula approximate the cost of a single pass over all the rows of R and S. The last term represents the size of the join result to be produced. Of course, the cost of output is the same for all join algorithms, but we did not need to include it in the nested loop algorithm cost estimation because it is smaller than the cost of nested loops.

This formula shows that a hash-based algorithm is significantly better than nested loops for large tables and a large number of different values of the join attribute. For example, if the join attribute is unique in one of the input tables, then the last term will be equal to just the size of the other table.

The basic hash join algorithm works if all buckets produced at the build phase can fit into main memory. Another variation, called hybrid hash join, joins tables that cannot fit into main memory. The hybrid hash join partitions both tables so that partitions of one table can fit and then executes a basic algorithm for each pair of corresponding partitions. The cost of a hybrid hash join is higher because partitions are stored temporarily on the hard disk and both tables are scanned twice. However, the cost is still proportional to the sum of the sizes, rather than the product.

Sort-Merge Algorithm

Another algorithm (called sort-merge) for equi-joins is schematically shown in Figure 3-8.

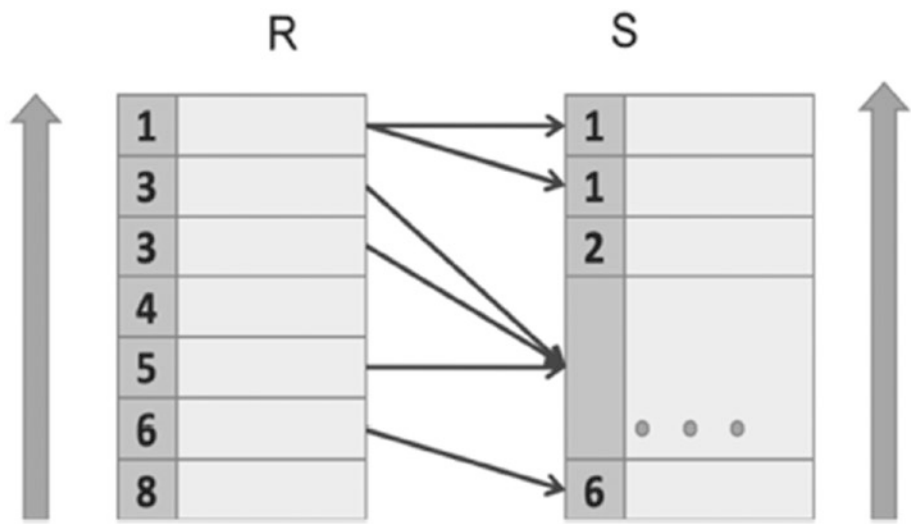


Figure 3-8. *Sort-merge algorithm*

The first phase of the algorithm sorts both input tables in ascending order by the join attribute.

When the input tables are properly ordered, the merge phase scans both input tables once and, for each value of the join attribute, calculates the Cartesian product of rows containing this value of the join attribute. Note that this product is a necessary part of the join result. New rows with the same value attribute cannot appear in the remaining part of input because the input tables are ordered.

The cost of the merge phase can be expressed with the same formula as for a hash join, that is, proportional to the sum of sizes of input and output. The actual cost is somewhat lower because there is no need for the build phase.

The cost of sorting can be estimated with the following formula:

$$\text{Size}(R) \cdot \log(\text{size}(R)) + \text{size}(s) \cdot \log(\text{size}(S))$$

The sort-merge algorithm is especially efficient if one of both input tables is already sorted. This may happen in a series of joins with the same join attribute.

Comparing Algorithms

Just as with data access algorithms, there are no default winners or losers. Any of the algorithms can be the best, depending on the circumstances. The nested loop algorithm is more universal and is the best for small index-based joins; a sort-merge and hash are more efficient for large tables, when applicable.

Summary

Having covered cost models for algorithms, data access algorithms, the purpose and structure of indexes, and algorithms for more complex operations, such as joins, we finally have enough building blocks to move on to the full product of the query planner—an execution plan.

The next chapter covers how to read and understand execution plans and improve them.

CHAPTER 4

Understanding Execution Plans

At long last, it's time to look at execution plans. Before we begin, let's review our theoretical foundations. Chapter 3 explained how logical operations are mapped to their physical execution, covering data retrieval and more complex operations.

In this chapter, understanding these algorithms will enable us to interpret execution plans and get a better grasp of their components.

Putting Everything Together: How an Optimizer Builds an Execution Plan

The output of the PostgreSQL optimizer is an *execution plan*. While a SELECT defines *what* needs to be done, an execution plan defines *how* to execute SQL operations.

The job of the optimizer is to build the best possible physical plan that implements a given logical plan. This is a complex process: sometimes, a complex logical operation is replaced with multiple physical operations, or several logical operations are merged into a single physical operation.

To build a plan, the optimizer uses *transformation rules*, *heuristics*, and cost-based *optimization algorithms*. A *rule* converts a plan into another plan with better cost. For example, filter and project operations reduce the size of the dataset and therefore should be executed as early as possible; a rule might reorder operations so that filter and project operations are executed sooner. An *optimization algorithm* chooses the plan with the lowest cost estimate. However, the number of possible plans (called the *plan space*) for a query containing several operations is huge—far too large for the algorithm to

consider every single possible plan. After all, time spent choosing the correct algorithm contributes to the total execution time of the query. *Heuristics* are used to reduce the number of plans evaluated by the optimizer.

Reading Execution Plans

To paraphrase Elvis, a little less abstraction, a little more action, please. Let's take a look at an example. The query in Listing 4-1 selects all flights that departed from JFK and arrived at ORD with a scheduled departure between August 10 and August 23, 2023. For each flight, the total number of passengers is calculated.

Listing 4-1. A query selecting the number of passengers on specific flights

```
SELECT f.flight_no,
       f.actual_departure,
       count(passenger_id) passengers
FROM flight f
JOIN booking_leg bl ON bl.flight_id = f.flight_id
JOIN passenger p ON p.booking_id=bl.booking_id
WHERE f.departure_airport = 'JFK'
      AND f.arrival_airport = 'ORD'
      AND f.actual_departure BETWEEN
          '2023-08-10' and '2023-08-13'
GROUP BY f.flight_id, f.actual_departure;
```

A logical plan for this query is shown in Listing 4-2.

Listing 4-2. The logical plan for the query in Listing 4-1

```
project f.flight_no, f.actual_departure, count(p.passenger_id)[] (
  group [f.flight_no, f.actual_departure] (
    filter [f.departure_airport = 'JFK'] (
      filter [f.arrival_airport = 'ORD'] (
        filter [f.actual_departure >='2023-08-10'] (
          filter [f.actual_departure <='2023-08-13' ] (
            join [bl.flight_id = f.flight_id] (
              access (flights f),
```



```

join(bl.booking_id=p.booking_id (
  access (booking_leg bl),
  access (passenger p)
))))))

```

The logical plan shows which logical operations should be executed, but it does not provide details on how they will be executed. The query planner produces an execution plan for the query, shown in Figure 4-1.

	QUERY PLAN text
1	GroupAggregate (cost=762880.24..762887.73 rows=4 width=24)
2	Group Key: f.flight_id
3	-> Sort (cost=762880.24..762882.72 rows=994 width=20)
4	Sort Key: f.flight_id
5	-> Hash Join (cost=366850.85..762830.75 rows=994 width=20)
6	Hash Cond: (p.booking_id = bl.booking_id)
7	-> Seq Scan on passenger p (cost=0.00..334798.06 rows=16312506 width=8)
8	-> Hash (cost=366849.54..366849.54 rows=105 width=20)
9	-> Hash Join (cost=9371.98..366849.54 rows=105 width=20)
10	Hash Cond: (bl.flight_id = f.flight_id)
11	-> Seq Scan on booking_leg bl (cost=0.00..310506.66 rows=17893566 width=8)
12	-> Hash (cost=9371.93..9371.93 rows=4 width=16)
13	-> Bitmap Heap Scan on flight f (cost=123.95..9371.93 rows=4 width=16)
14	Recheck Cond: (departure_airport = 'JFK':bpchar)
15	Filter: ((actual_departure >= '2023-08-10 00:00:00-05':timestamp with time zone) AND (actual_departure <= '2023-08-13 00:00:00-05':timestamp with time zone))
16	-> Bitmap Index Scan on flight_departure_airport (cost=0.00..123.94 rows=11136 width=0)
17	Index Cond: (departure_airport = 'JFK':bpchar)

Figure 4-1. Execution plan

To obtain the execution plan for a query, the EXPLAIN command is run. This command takes any grammatically correct SQL statement as a parameter and returns its execution plan.

We encourage you to run the code examples throughout this book and examine the execution plans. However, a word of caution: Choosing the correct execution plan is a nondeterministic process. The plans that your local database produces might differ slightly from the plans shown in this book. Even when the plans are identical, execution times may vary with differences in hardware and configuration.

Hopefully, looking at Figure 4-1, the value of the preceding chapters is evident—each line represents an operation previously covered, so it's clear what's going on under the hood. Note that, in addition to the names of the algorithms, each line of the execution plan includes several mysterious numbers in parentheses. This mystery can be easily resolved by recalling Chapter 3, which discussed how the costs of different algorithms are calculated.

Specifically, a plan contains the expected number of output rows and the expected average width of output row, which are calculated from database statistics, and estimates of costs, which are calculated based on the previously obtained estimates and configuration parameters. The values of costs include the accumulated cost of all previous operations. There are two cost estimations for each operation: the first shows the cost needed to produce the first row of output, while the second estimates the cost of the complete result. For example, in Figure 4-1, line 13, 123.95 is the cost of the first row of output and 9371.93 the cost of the complete result. Estimates for the number and width of output rows are needed to estimate the cost of an operation that consumes the output. How costs are estimated will be covered later in this chapter.

It is important to emphasize that all these numbers are approximate. The actual values obtained during execution may differ. If you suspect that the optimizer chose a plan that is not optimal, you might need to look at these estimates. Usually, the error is small for stored tables, but it inevitably accumulates after each operation.

An execution plan is presented as a tree of physical operations. In this tree, nodes represent operations, and arrows point to operands. The tree structure might not be apparent from looking at Figure 4-1. There are multiple tools, including pgAdmin, which can generate a graphical representation of execution plans. Figure 4-2 illustrates what an output might look like. (Different graphic tools may use different iconography; this example is presented for illustrative purposes only.) In fact, this figure represents the execution plan for Listing 4-4, which we discuss later in this chapter.

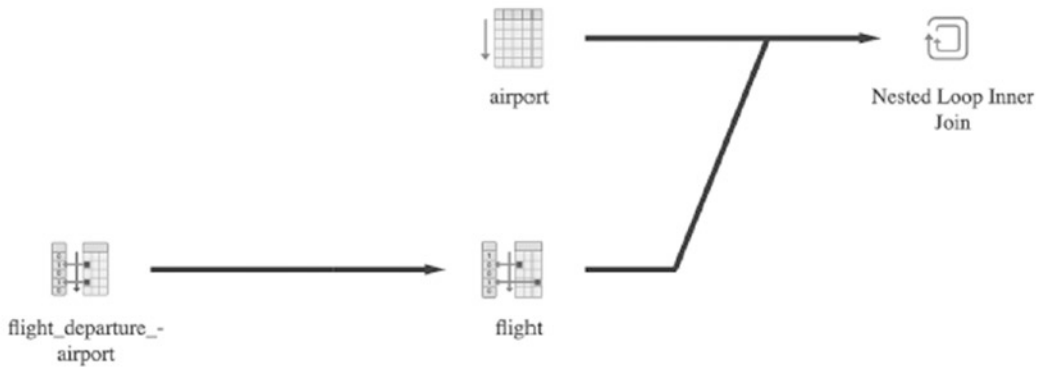


Figure 4-2. Graphical representation of a simple execution plan (Listing 4-4)

For more complex queries, the graphical representation of the execution plan may be less helpful—see the graphical representation of the execution plan for Listing 4-1 in Figure 4-3.

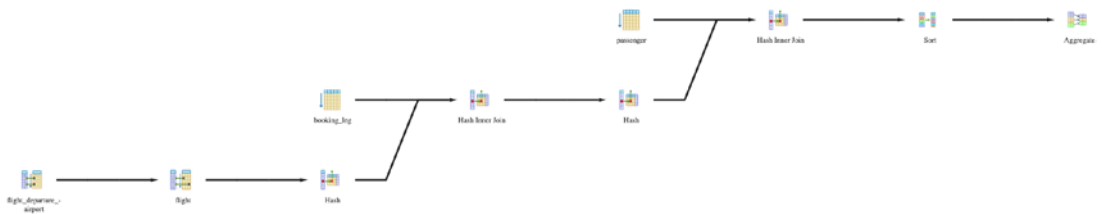


Figure 4-3. Graphical representation of the execution plan for Listing 4-1

In such cases, a more compact representation could be more useful, like the one presented in Figure 4-4.

	Node
1.	→ Aggregate
2.	→ Sort
3.	→ Hash Inner Join Hash Cond: (p.booking_id = bl.booking_id)
4.	→ Seq Scan on passenger as p
5.	→ Hash
6.	→ Hash Inner Join Hash Cond: (bl.flight_id = f.flight_id)
7.	→ Seq Scan on booking_leg as bl
8.	→ Hash
9.	→ Bitmap Heap Scan on flight as f Filter: ((actual_departure >= '2023-08-10 00:00:00-05':timestamp with time zone) AND (actual_departure <= '2023-08-10 23:59:59-05':timestamp with time zone)) Recheck Cond: (departure_airport = 'JFK':bpchar)
10.	→ Bitmap Index Scan using flight_departure_airport Index Cond: (departure_airport = 'JFK':bpchar)

Figure 4-4. Alternative representation of the same execution plan

Now, let’s get back to the actual output of the EXPLAIN command, shown in Figure 4-1. It shows each node of the tree on a separate line starting with ->, with the depth of the node represented by the offset. Subtrees are placed after their parent node. Some operations are represented with two lines.

The execution of a plan starts from the leaves and ends at the root. This means that the operation that is executed first will be on the line that has the rightmost offset. Of course, a plan may contain several leaf nodes that are executed independently. As soon as an operation produces an output row, this row is pushed up to the next operation. Thus, there is no need to store intermediate results between operations.

In Figure 4-1, execution starts from the last line, accessing the table flight using the index on the departure_airport column. Since several filters are applied to the table and only one of the filtering conditions is supported by the index, PostgreSQL performs a bitmap index scan (covered in Chapter 2). The engine accesses the index and compiles the list of blocks that could contain needed records. Then, it reads the actual blocks from the database using bitmap heap scan, and for each record extracted from the database, it rechecks that rows found via the index are current and applies filter operations for additional conditions for which we do not have indexes: arrival_airport and scheduled_departure.

The result is joined with the table `booking_leg`. PostgreSQL uses a sequential read to access this table and a hash join algorithm on the condition `b1.flight_id = f.flight_id`.

Then, the table `passenger` is accessed via a sequential scan (since it doesn't have any indexes), and once again, the hash join algorithm is used on the `p.booking_id = b1.booking_id` condition.

The last operation to be executed is grouping and calculating the aggregate function `sum()`. Sorting determines the flights that satisfy the search criteria. Then, the count of all passengers on each of the flights is performed.

The next section addresses what else can be gleaned from the execution plan and why it is important.

Understanding Execution Plans

Understanding small execution plans is easy. Recall that in Chapter 3 we compared two `SELECT` statements, Listings 3-2 and 3-3. We stated that for the first one, PostgreSQL would choose a sequential scan, while for the second one it would choose index access. How did we know that? Now, you know the answer: we ran the `EXPLAIN` command for both statements. The execution plan for Listing 3-2 is presented in Figure 4-5 and the execution plan for Listing 3-3 in Figure 4-6.

	QUERY PLAN text
1	Seq Scan on flight (cost=0.00..18814.67 rows=353801 width=12)
2	Filter: ((scheduled_departure >= '2023-05-15 00:00:00-05':timestamp with time zone) AND (scheduled_departure <= '2023-08-31 00:00:00-05':timestamp with time zone))

Figure 4-5. Execution plan for Listing 3-3

	QUERY PLAN text
1	Bitmap Heap Scan on flight (cost=51.87..6611.44 rows=3458 width=12)
2	Recheck Cond: ((scheduled_departure >= '2023-08-12 00:00:00-05':timestamp with time zone) AND (scheduled_departure <= '2023-08-13 00:00:00-05':timestamp with time zone))
3	-> Bitmap Index Scan on flight_scheduled_departure (cost=0.00..51.00 rows=3458 width=0)
4	Index Cond: ((scheduled_departure >= '2023-08-12 00:00:00-05':timestamp with time zone) AND (scheduled_departure <= '2023-08-13 00:00:00-05':timestamp with time zone))

Figure 4-6. Execution plan for Listing 3-4

But what is an execution plan is more complex?

Often, when we explain how to read execution plans in the manner described in the preceding text, our audience feels overwhelmed by the size of the execution plan for a relatively simple query, especially given that a more complex query can produce an execution plan of 100+ lines. Even the plan presented in Figure 4-1 might require some time to read. Sometimes, even when each and every single line of a plan can be interpreted, the question remains: “I have a query, and it is slow, and you tell me to look at the execution plan, and it is 100+ lines long. What should I do? Where should I start?”

The good news is that most of the time, you do not need to read the whole plan to understand what exactly makes the execution slow. In this section, we will learn more about interpreting execution plans.

What Is Going On During Optimization?

As mentioned in Chapter 2, the optimizer performs two kinds of transformations: it replaces logical operations with corresponding physical execution algorithms and (possibly) changes the logical expression structure by changing the order in which logical operations are executed.

After the query is parsed and checked for syntactic correctness, the first step is query rewriting. In this step, the PostgreSQL optimizer enhances the code by eliminating subqueries, substituting views with their textual presentation, and so on. It is essential to keep in mind that this step always happens. When the concept of a view is introduced, SQL textbooks often suggest that “views can be used like tables,” which is misleading. In most cases, views are substituted by their source code. However, “most of the time” does not mean “always.” Chapter 7 discusses views, how the optimizer processes them, and their potential performance pitfalls.

The next step after query rewrite is what we usually call optimization, which includes the following:

- Determining the possible orders of operations
- Determining the possible execution algorithms for each operation
- Comparing the costs of different plans
- Selecting the optimal execution plan

Many SQL developers presume that PostgreSQL executes queries accessing (and joining) tables in the same order they appear in the FROM clause.

However, the order of joins is *not* preserved most of the time—the database does not expect these instructions. In subsequent chapters, we will discuss in more detail what influences the order of operations. For now, let’s consider how to evaluate an execution plan.

Why Are There So Many Execution Plans to Choose From?

We’ve noted several times that one SQL statement can be executed in many ways, using different execution plans. In fact, there could be hundreds, thousands, or even millions of possible ways to execute one statement! This chapter gives some sense of where these numbers are coming from. Plans may vary in

- Order of operations
- Algorithms used for joins and other operations (e.g., nested loops, hash join)
- Data retrieval methods (e.g., index usage, full scan)

Formally speaking, the optimizer finds the best plan by computing the costs for all possible plans and then comparing the costs. But since we know that there are three basic algorithms to execute each join, even a simple SELECT on three tables can generate nine possible execution plans; given the 12 possible join orders, there are 108 possible plans ($3 * 3 * 12 = 108$). If we then consider all the potential data retrieval methods for each table, there are several thousand plans to compare.

Fortunately, PostgreSQL does not check every possible plan.

The cost-based optimization algorithm relies on the optimality principle: a sub-plan of an optimal plan is optimal for the corresponding subquery. A plan can be considered a composition of multiple component pieces, or sub-plans. A sub-plan is a plan that includes any operations of the original plan as a root node and all its descendant nodes, that is, all operations that contribute to the input arguments for the operation chosen as a root of the sub-plan. The optimizer builds the optimal plan starting from the smallest sub-plans (i.e., data access to single tables) and gradually produces more complex sub-plans, including more operations with only a few checks of cost on each step. The algorithm is exhaustive in the sense that the optimal plan will be built, despite the fact that a significant portion of possible plans will not be tried.

For example, in the preceding example, once the optimizer selects the correct data retrieval algorithm for one of the three tables, it will not consider any plans that do not use this optimal algorithm.

Still, the number of produced sub-plans can be huge. The `geqo_threshold` configuration parameter specifies the maximum number of joins in a query for which the near-exhaustive search of the best join sequence is performed. If the number of tables exceeds the maximum, the joins order is determined by heuristics. Heuristics cut out parts of the plan space that are unlikely to contain optimal plans, reducing the number of plans examined by the optimization algorithm. While this feature helps the optimizer select an execution plan more quickly, it can also affect performance negatively: there is a risk that the best execution plan will be accidentally dropped before the cost comparison.

Although heuristics may cut out the optimal plan, the algorithm builds the best of the remaining plans.

Now, let's take a closer look at how these costs are calculated.

How Are Execution Costs Calculated?

In Chapter 3, we discussed ways to measure the performance of database algorithms. We talked about internal metrics and established that the costs of algorithms are measured in the number of I/O operations and CPU cycles. Now, we are going to apply this theory to practice.

The cost of each execution plan depends on

- Cost formulas of algorithms used in the plan
- Statistical data on tables and indexes, including distribution of values
- System settings (parameters and preferences), such as `join_collapse_limit` or `cpu_index_tuple_cost`

Chapter 3 covered the formulas to calculate cost for each algorithm. Each of these formulas depends on the size of the table(s) used, as well as on the expected size of the result set. And finally, users can alter the default cost for operations with system settings. The choice of an optimal plan can be implicitly controlled by changes in the optimizer parameters that are used during the cost estimation. Thus, all three pieces of information factor into the calculation of the cost of execution plans.

This is counterintuitive; often, SQL developers have the subconscious expectation that the “best possible plan” exists and, moreover, that it is the same for all “similar” queries. However, due to the factors listed in the preceding list, the optimizer may produce different execution plans for nearly identical SQL queries or even for the same query. How can this happen? The optimizer chooses the plan with the best cost estimation. However, there may be several plans with only slightly different costs. The cost estimation depends on the database statistics that are gathered from random samples. The statistics gathered yesterday may slightly differ from those gathered today. Due to these slight changes, a plan that was the best yesterday can become second best today. Of course, statistics may also change as a result of insertions, updates, and deletions.

Let’s look at some examples. Listings 4-3 and 4-4 present two queries, which appear almost identical. The only difference is in the filtering value. However, the execution plans presented in Figures 4-7 and 4-8 are markedly different.

Listing 4-3. Simple SELECT with one condition

```
SELECT flight_id,
       scheduled_departure
FROM flight f
JOIN airport a ON departure_airport=airport_code
               AND iso_country='US'
```

	QUERY PLAN
	text
1	Hash Join (cost=20.09..25467.60 rows=144636 width=12)
2	Hash Cond: (f.departure_airport = a.airport_code)
3	-> Seq Scan on flight f (cost=0.00..23642.76 rows=683176 width=16)
4	-> Hash (cost=18.33..18.33 rows=141 width=4)
5	-> Seq Scan on airport a (cost=0.00..18.33 rows=141 width=4)
6	Filter: (iso_country = 'US'::text)

Figure 4-7. Execution plan for Listing 4-3

Listing 4-4. The same SELECT as Listing 4-3, with a different search value

```
SELECT flight_id,  
       scheduled_departure  
FROM flight f  
JOIN airport a ON departure_airport=airport_code  
                AND iso_country='CZ'
```

QUERY PLAN		text	
1	Nested Loop	(cost=40.40..3347.29 rows=1026 width=12)	
2	-> Seq Scan on airport a	(cost=0.00..18.33 rows=1 width=4)	
3	Filter:	(iso_country = 'CZ'::text)	
4	-> Bitmap Heap Scan on flight f	(cost=40.40..3318.67 rows=1029 width=16)	
5	Recheck Cond:	(departure_airport = a.airport_code)	
6	-> Bitmap Index Scan on flight_departure_airport	(cost=0.00..40.14 rows=1029 width=0)	
7	Index Cond:	(departure_airport = a.airport_code)	

Figure 4-8. Execution plan for Listing 4-4

What causes this difference? Figure 4-9 gives a clue: The first query selects a significant portion of all airports, and using an index won't improve performance. The second query, by contrast, will select only one airport, and in this case index-based access will be more efficient.

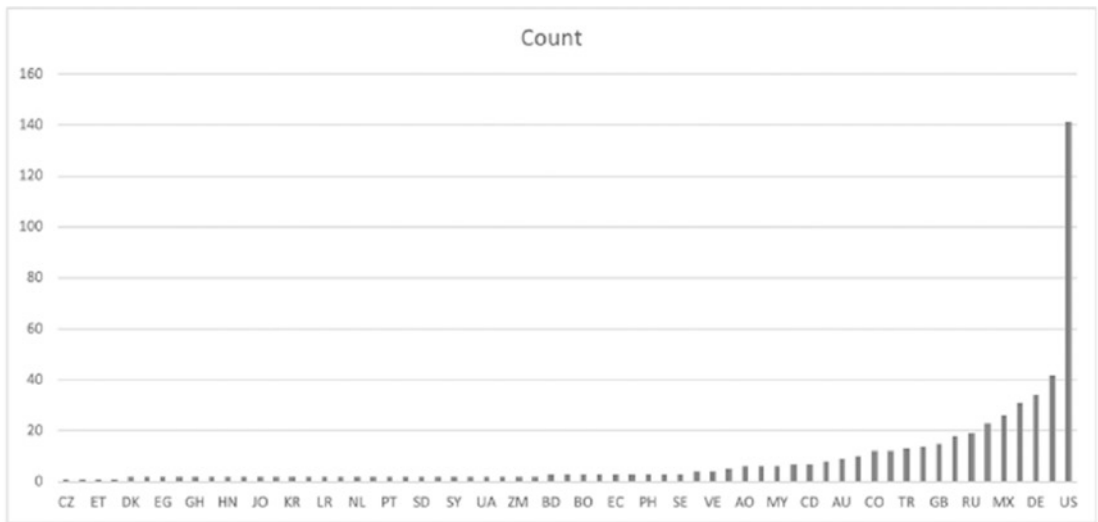


Figure 4-9. *A histogram of value distribution*

This example also explains why keeping the database statistics up to date is so important. If the query planner won't be aware of such uneven distribution, it might choose an execution plan that would be far from optimal. The `ANALYZE` command recalculates statistics for a table that is passed as a parameter and updates statistics on the table's indexes. PostgreSQL has a special mechanism to update statistics automatically, and we will discuss it in more detail in Chapter 8. However, if you are following the examples from this book and replay them on your local copy, you should manually run `ANALYZE` after building new indexes.

How Can the Optimizer Be Led Astray?

But how can we be sure that the plan the optimizer has selected is indeed the best possible plan? Is it even possible to find the best execution plan? We spent quite a bit of time explaining that the optimizer does the best possible job if we leave it alone and don't interfere. If that's true, what is the rest of the book about? The reality is that no optimizer is perfect, even the PostgreSQL query planner.

First, while the optimization algorithm is mathematically correct—it finds the plan with the best cost estimation—those cost estimates are intrinsically imprecise. The simple formulas explained in Chapter 3 are valid only for a uniform distribution of

data, but a uniform distribution seldom occurs in real databases. In reality, optimizers use more complex formulas, but these are also imperfect approximations of reality. As George Box said, “All models are wrong, but some are useful.”

Second, database systems, including PostgreSQL, maintain detailed statistics of stored data (usually in histograms). Histograms significantly improve estimates of selectivity. Unfortunately, histograms cannot be used for intermediate results. Errors in estimating intermediate results are the primary reason the optimizer may fail to produce an optimal plan.

Third, an optimal plan may be cut out with heuristics, or a query might be too complex for an exact optimization algorithm. In the latter case, approximate optimization algorithms are used.

In all these cases, some human intervention is required, and that’s what this book is about! Now that we know what’s going on during optimization, we can fix it if something doesn’t work quite right.

In spite of these potential hiccups, optimizers work well in the majority of cases. However, humans observe the behavior of the system and therefore have more information available than the optimizer and can use this additional knowledge to help the optimizer do its job even better.

Summary

This chapter covered execution plans: how they are generated and how to read and understand them. We also learned about cost-based optimization and factors that impact the cost of execution plans.

Although cost-based optimizers usually do a good job, sometimes they need help, and now we are well equipped to provide it. Subsequent chapters will go over multiple examples of queries that require some human intervention to achieve better performance.

CHAPTER 5

Short Queries and Indexes

Chapter 4 took a deep dive into understanding execution plans. Now, we turn to what to do once the EXPLAIN command has been run and an execution plan is returned. Where do we start if our goal is to improve the query execution plan?

The first step is to identify whether the query is a short query or a long query. This chapter focuses on optimizing short queries. You will learn how to identify short queries, what optimization technique to use with short queries, and why indexes are critically important for this query type. We also discuss different types of indexes available in PostgreSQL and when each index type can be used.

Before proceeding with this chapter, let's create several additional indexes:

```
SET search_path TO postgres_air;  
CREATE INDEX flight_arrival_airport ON flight (arrival_airport);  
CREATE INDEX booking_leg_flight_id ON booking_leg (flight_id);  
CREATE INDEX flight_actual_departure ON flight (actual_departure);  
CREATE INDEX boarding_pass_booking_leg_id ON boarding_pass (booking_  
leg_id);  
CREATE INDEX booking_update_ts ON booking (update_ts);
```

Don't forget to run ANALYZE on all the tables for which you build new indexes:

```
ANALYZE flight ;  
ANALYZE booking_leg;  
ANALYZE booking;  
ANALYZE boarding_pass;
```

What Makes a Query “Short”?

The term *short query* has come up multiple times, without a formal definition. What is a short query? First, it has nothing to do with the length of the SQL query. Take a look at the two queries shown in Listings 5-1 and 5-2, for example. The query in Listing 5-1 contains only four lines of code, but it represents a *long query*. Listing 5-2 contains many more lines but is a short query.

Listing 5-1. Long query example

```
SELECT
    d.airport_code AS departure_airport,
    a.airport_code AS arrival_airport
FROM  airport a,
      airport d
```

Listing 5-2. Short query example

```
SELECT
    f.flight_no,
    f.scheduled_departure,
    boarding_time,
    p.last_name,
    p.first_name,
    bp.update_ts as pass_issued,
    ff.level
FROM  flight f
      JOIN booking_leg bl ON bl.flight_id = f.flight_id
      JOIN passenger p ON p.booking_id=bl.booking_id
      JOIN account a on a.account_id =p.account_id
      JOIN boarding_pass bp on bp.passenger_id=p.passenger_id
      LEFT OUTER JOIN frequent_flyer ff on ff.frequent_flyer_id=a.frequent_
      flyer_id
WHERE f.departure_airport = 'JFK'
      AND f.arrival_airport = 'ORD'
```

```
AND f.scheduled_departure BETWEEN
      '2023-08-05' AND '2023-08-07'
```

Second, it is not defined by the size of the result set. The query in Listing 5-3 yields only one line; however, it is a long query.

Listing 5-3. Long query that produces one row

```
SELECT
    avg(flight_length),
    avg (passengers)
FROM (SELECT
        flight_no,
        scheduled_arrival -scheduled_departure AS flight_length,
        count(passenger_id) passengers
    FROM flight f
    JOIN booking_leg bl ON bl.flight_id = f.flight_id
    JOIN passenger p ON p.booking_id=bl.booking_id
    GROUP BY 1,2) a
```

So what is a short query?

A query is short when the number of rows needed to compute its output is small, no matter how large the involved tables are. Short queries may read every row from small tables but read only a small percentage of rows from large tables.

The most important part of the preceding definition is the difference between *needs* and *accesses*. A poor execution plan for a short query may read and/or process many more rows than needed for final output. Although a certain amount of redundant processing is unavoidable, our goal is to reduce it.

How small is a “small percentage”? Unsurprisingly, it depends on system parameters, application specifics, actual table sizes, and possibly other factors. Most of the time, however, it means less than 10%. Later in this chapter, a case study will show how to identify this border line.

By contrast, the output of a long query depends on a significant fraction of rows in a large table or several large tables.

Our taxonomy of queries is similar to the commonly accepted distinction between OLTP and OLAP queries. All OLTP queries are short. However, many modern applications require queries that return hundreds of rows but still are short.

Why is Listing 5-1 a long query? Because all the rows from the airport table are required to obtain the result. Why is Listing 5-2 a short query? Because data from just a couple of flights is needed, out of about 200,000. Why isn't Listing 5-3 short? Because data from every booking in the system is required to calculate the results.

When we optimize a short query, we know that in the end, we select a relatively small number of records. This means that the optimization goal is to reduce the size of the result set as early as possible. If the most restrictive selection criterion is applied in the first steps of query execution, further sorting, grouping, and even joins will be less expensive. Looking at the execution plan, there should be no table scans of large tables. For small tables, a full scan may still work, as shown in Figure 3-2 in Chapter 3.

Choosing Selection Criteria

It might seem easy to make sure that the most restrictive selection criteria are applied first; however, this isn't always straightforward. To acknowledge the obvious, this chapter is called "Short Queries and Indexes" for a reason: you can't select a subset of records quickly from a table if there is no index supporting the corresponding search. That's why short queries require indexes for faster execution. If there is no index to support a highly restrictive query, in all likelihood, one needs to be created.

Index Selectivity

Chapter 3 introduced the concept of *query selectivity*. The same concept can be applied to indexes: the smaller the number of records that correspond to one value of the index, the lower the index's selectivity value. We do not want to create indexes with high selectivity; as we saw in Chapter 3, index-based data retrieval in this case will take *more* time than a sequential scan. Since the PostgreSQL optimizer predetermines the cost of each access method, this index would never be used, so performance wouldn't be compromised. However, it is still undesirable to add a database object that requires storage space and extra time to update, but doesn't provide any benefit.

A database table might have multiple indexes on different columns, each with different selectivity. The best performance possible for a short query occurs when the most restrictive indexes (i.e., indexes with the lowest selectivity) are used.

Let's look at the query in Listing 5-4. Can you tell which filtering criterion is the most restrictive?

Listing 5-4. Index selectivity

```
SELECT *
FROM flight
WHERE departure_airport='LAX'
      AND update_ts BETWEEN '2023-08-13' AND '2023-08-15'
      AND status='Delayed'
      AND scheduled_departure BETWEEN '2023-08-13' AND
      '2023-08-15'
```

Delayed status might be the most restrictive, because ideally, on any given day, there are many more on-time flights than delayed flights.

In our training database, we have a flight schedule for six months, so limiting it by two days might not be very restrictive. On the other hand, usually the flight schedule is posted well in advance, and if we are looking for flights where the timestamp of the last update is relatively close to the scheduled departure, it most likely indicates that these flights were delayed or canceled.

Another factor that may be taken into consideration is the popularity of the airport in question. LAX is a popular airport, and for Listing 5-4, a restriction on `update_ts` will be more restrictive than on `departure_airport`. However, if we change the filtering on `departure_airport` to FUK, the airport criterion will be more restrictive than selection based on `update_ts`.

If all the search criteria are indexed, there is no cause for concern; the way multiple indexes can work together will be covered in a moment. But if the most restrictive criterion is not indexed, the execution plan may be suboptimal, and likely, an additional index is needed.

Unique Indexes and Constraints

The better (lower) the selectivity of an index, the faster the search. Thus, the most efficient indexes are unique indexes.

An index is *unique* if for each indexed value there is exactly one matching row in the table.

There are several different ways to create a unique index. First, PostgreSQL automatically creates a unique index to support any primary key or unique constraint on a table.

What is the difference between a primary key and a unique constraint? A common misconception among SQL developers is that a primary key has to be an incrementing numeric value and that each table “has” to have a primary key. Although it often helps to have a numeric incremental primary key (called a *surrogate key*), a primary key does not have to be numeric, and moreover, it does not have to be a single-attribute constraint. It is possible to define a primary key as a combination of several attributes; it just has to satisfy two conditions: the combination must be `UNIQUE` and `NOT NULL` for all of the participating attributes. In contrast, unique constraints in PostgreSQL allow for `NULL` values.

A table can have a single primary key (though a primary key is not required) and multiple unique constraints. Any non-null unique constraint can be chosen to be a primary key for a table; thus, there is no programmatic way to determine the best candidate for a table’s primary key. For example, the table `booking` has a primary key on `booking_id` and a unique key on `booking_ref`—see Listing 5-5.

Listing 5-5. A primary key and a unique constraint

```
ALTER TABLE booking
    ADD CONSTRAINT booking_pkey PRIMARY KEY (booking_id);
ALTER TABLE booking
    ADD CONSTRAINT booking_booking_ref_key UNIQUE (booking_ref);
```

Since `booking_ref` is a non-null attribute, we could choose either `booking_id` or `booking_ref` to be the primary key.

As shown in the ER diagram in Chapter 1, the column `frequent_flyer_id` in the table `account` is nullable and also unique:

```
ALTER TABLE account
  ADD CONSTRAINT account_freq_flyer_unq_key UNIQUE (frequent_flyer_id);
```

It is also possible to create a unique index without formally defining a unique constraint. All you have to do is to add the keyword `unique` to the index creation statement:

```
CREATE UNIQUE INDEX account_freq_flyer ON
account (frequent_flyer_id);
```

If we create this index after data was already inserted into this table, `CREATE UNIQUE INDEX` will validate the uniqueness of values, and if any duplicates are found, the index won't be created. For any subsequent inserts and updates, the uniqueness of new values will be validated as well.

What about foreign keys? Do they automatically create any indexes? A common misconception is the belief that the presence of a foreign key necessarily implies the presence of an index on the child table. This is not true.

A foreign key is a referential integrity constraint; it guarantees that for each non-null value in the child table (i.e., the table with the foreign key constraint), there is a matching unique value in the parent table (i.e., the table it is referencing).

For example, there is a foreign key constraint on the `flight` table that ensures that each arrival airport matches an existing airport code:

```
ALTER TABLE flight
  ADD CONSTRAINT arrival_airport_fk FOREIGN KEY (arrival_airport)
  REFERENCES airport (airport_code);
```

This constraint does not automatically create an index; if searches by arrival airport are slow, the index must be explicitly created:

```
CREATE INDEX flight_arrival_airport
  ON flight (arrival_airport);
```

Chapter 3 mentioned that unique indexes make nested loops efficient. If you refer to Figure 3-7, you will realize what happens when an index is present.

The nested loop join algorithm can also be combined with index-based data access if the table *S* has an index on attributes used in the join condition. For natural joins, the inner loop of the index-based nested loop algorithm shrinks to few rows of *S* for each row of *R*. The inner loop can even vanish completely if the index on *S* is unique, for example, the join attribute of *S* is its primary key.

Often, this is misunderstood to mean that nested loops are always efficient when joining on a primary/foreign key. However, as mentioned earlier, this is true only if the column in the child table—that is, the foreign key—is indexed.

Is it a best practice to always create an index on a column that has a foreign key constraint? Not always. An index should only be created if the number of distinct values is large enough. Remember, indexes with high selectivity are unlikely to be useful. For example, the *flight* table has a foreign key constraint on *aircraft_code*:

```
ALTER TABLE flight
  ADD CONSTRAINT aircraft_code_fk FOREIGN KEY (aircraft_code)
  REFERENCES aircraft (code);
```

This foreign key constraint is necessary because for each flight, there must be a valid aircraft assigned. In order to support the foreign key constraint, a primary key constraint was added to the *aircraft* table. That table, however, has only 12 rows. Therefore, it is not necessary to create an index on the *aircraft_code* column of the *flight* table. This column has only 12 distinct values, so an index on that column will not be used.

To illustrate this statement, let's look at the query in Listing 5-6. This query selects all flights departing from JFK airport between August 14 and 16, 2023. For each flight, we select the flight number, scheduled departure, aircraft model, and number of passengers.

Listing 5-6. A join by a primary/foreign key without an index

```
SELECT
  f.flight_no,
  f.scheduled_departure,
  model,
  count(passenger_id) passengers
FROM flight f
JOIN booking_leg bl ON bl.flight_id = f.flight_id
```

```

JOIN passenger p ON p.booking_id=bl.booking_id
JOIN aircraft ac ON ac.code=f.aircraft_code
WHERE f.departure_airport = 'JFK'
      AND f.scheduled_departure BETWEEN
      '2023-08-14' AND '2023-08-16'
GROUP BY 1,2,3

```

The execution plan for this query is shown in Figure 5-1, and it is massive.

	QUERY PLAN text
1	GroupAggregate (cost=398695.75..398727.72 rows=1260 width=52)
2	Group Key: f.flight_no, f.scheduled_departure, ac.model
3	-> Sort (cost=398695.75..398699.62 rows=1550 width=48)
4	Sort Key: f.flight_no, f.scheduled_departure, ac.model
5	-> Hash Join (cost=2615.33..398613.61 rows=1550 width=48)
6	Hash Cond: (p.booking_id = bl.booking_id)
7	-> Seq Scan on passenger p (cost=0.00..334807.39 rows=16313439 width=8)
8	-> Hash (cost=2613.26..2613.26 rows=165 width=48)
9	-> Nested Loop (cost=222.22..2613.26 rows=165 width=48)
10	-> Hash Join (cost=221.78..609.05 rows=6 width=48)
11	Hash Cond: ((f.aircraft_code)::text = ac.code)
12	-> Bitmap Heap Scan on flight f (cost=220.51..607.47 rows=105 width=20)
13	Recheck Cond: ((scheduled_departure >= '2023-08-13 00:00:00-05':timestamp with time zone) AND (scheduled_departure <= '2023-08-16 00:00:00-05':timestamp with time zone))
14	-> BitmapAnd (cost=220.51..220.51 rows=105 width=0)
15	-> Bitmap Index Scan on flight_scheduled_departure (cost=0.00..100.88 rows=6845 width=0)
16	Index Cond: ((scheduled_departure >= '2023-08-13 00:00:00-05':timestamp with time zone) AND (scheduled_departure <= '2023-08-16 00:00:00-05':timestamp with time zone))
17	-> Bitmap Index Scan on flight_departure_airport (cost=0.00..119.33 rows=10521 width=0)
18	Index Cond: (departure_airport = 'JFK'::bpchar)
19	-> Hash (cost=1.12..1.12 rows=12 width=64)
20	-> Seq Scan on aircraft ac (cost=0.00..1.12 rows=12 width=64)
21	-> Index Scan using booking_leg_flight_id on booking_leg bl (cost=0.44..333.19 rows=85 width=8)
22	Index Cond: (flight_id = f.flight_id)

Figure 5-1. A plan with a sequential scan of a small table

The only part of this plan we are interested in now is this:

```

Hash (cost=1.12..1.12 rows=12 width=64)
-> Seq Scan on aircraft ac (cost=0.00..1.12 rows=12

```

The PostgreSQL optimizer accesses table statistics and is able to detect that the size of the aircraft table is small and index access won't be efficient. In contrast, as we observed earlier in this chapter, the index on the departure_airport field of the flight table proved to be useful due to its low selectivity.

Indexes and Non-equal Conditions

Chapter 3 described the structure of B-tree indexes, how they are built, and how they are used for searches. What follows is a demonstration of their practical application.

The previous section relates to simple B-tree indexes. As noted in Chapter 3, they can support searches by equality, greater than, less than, and between conditions: all the searches that require comparison and ordering. The majority of searches in an OLTP system fall into this category, but there are also a nontrivial number of cases when search criteria are more complex.

Indexes and Column Transformations

What is a column transformation? A column transformation occurs when the search criteria are on some modifications of the values in a column. For example, `lower(last_name)` (converting the `last_name` value to lowercase) and `update_ts::date` (casting timestamp with time zone to date) are column transformations.

How do column transformations affect index use? Quite simply, B-tree indexes on the attribute cannot be used. Recall from Chapter 3 how a B-tree is built and how a search on a B-tree is performed: in each node, the value of the attribute is compared to the value in the node. The transformed value is not recorded anywhere, so there is nothing to compare it to. Thus, if there is an index on last name

```
CREATE INDEX account_last_name
ON account (last_name);
```

...the following search won't be able to take advantage of the index:

```
SELECT *
FROM account
WHERE lower(last_name)='daniels';
```

How do we solve this problem? A search like this might be needed because passengers may enter their last names with different cases. If you believe that covering the most common cases is sufficient, you could modify the search criterion like so:

```
SELECT *
FROM account
WHERE last_name='daniels'
```

```
OR last_name='Daniels'
OR last_name = 'DANIELS'
```

The execution plan for this query is shown in Figure 5-2.

QUERY PLAN	
	text
1	Bitmap Heap Scan on account (cost=14.45..506.31 rows=143 width=53)
2	Recheck Cond: ((last_name = 'daniels'::text) OR (last_name = 'Daniels'::text) OR (last_name = 'DANIELS'::text))
3	-> BitmapOr (cost=14.45..14.45 rows=143 width=0)
4	-> Bitmap Index Scan on account_last_name (cost=0.00..4.78 rows=48 width=0)
5	Index Cond: (last_name = 'daniels'::text)
6	-> Bitmap Index Scan on account_last_name (cost=0.00..4.78 rows=48 width=0)
7	Index Cond: (last_name = 'Daniels'::text)
8	-> Bitmap Index Scan on account_last_name (cost=0.00..4.78 rows=48 width=0)
9	Index Cond: (last_name = 'DANIELS'::text)

Figure 5-2. An execution plan with “like” operator rewrite

A better solution would be to create an (additional) *functional* index:

```
CREATE INDEX account_last_name_lower
ON account (lower(last_name));
```

When a functional index is built, PostgreSQL applies the function to the values of the column (or columns) and then places these values in the B-tree. Similar to a regular B-tree index, where the nodes contain the values of the column, in a functional index, a node contains the value of the function. In our case, the function is `lower()`. After the index is created, query #1 in Listing 5-7 won’t use a sequential scan but will be able to utilize the new index. The corresponding execution plan is shown in Figure 5-3.

Listing 5-7. Different search conditions use different indexes

```
---#1
SELECT * FROM account WHERE lower(last_name)='daniels';
---#2
SELECT * FROM account WHERE last_name='Daniels';
---#3
SELECT * FROM account WHERE last_name='daniels';
---#4
SELECT * FROM account WHERE lower(last_name)='Daniels';
```

Note that an index on the `last_name` column is still necessary if we want a search on a case-specific value to be supported by an index (e.g., query #2). Also, it's worth mentioning that if the table `account` contains one record with `last_name = 'Daniels'` and another record with `last_name = 'DANIELS'`, query #1 will return both, query #2 will return only the first record, and queries #3 and #4 won't return either of them.

	QUERY PLAN
	text
1	Bitmap Heap Scan on account (cost=5.25..382.20 rows=107 width=53)
2	Recheck Cond: (lower(last_name) = 'daniels':text)
3	-> Bitmap Index Scan on account_last_name_lower_pattern (cost=0.00..5.22 rows=107 width=0)
4	Index Cond: (lower(last_name) = 'daniels':text)

Figure 5-3. A plan that uses a functional index

Note Sometimes, an additional index is not needed.

Should a functional index be created every time we need to search using a column transformation? Not necessarily. However, it is important to recognize a column transformation, which can be subtle.

For example, let's look at the following `SELECT` statement:

```
SELECT *
FROM flight
WHERE scheduled_departure ::date
BETWEEN '2023-08-17' AND '2023-08-18'
```

At first glance, it appears we are using the column `scheduled_departure` as a selection criterion, and since there is an index on this column, it should be used. However, the plan in Figure 5-4 diverts to a sequential scan.

	QUERY PLAN
	text
1	Seq Scan on flight (cost=0.00..22230.56 rows=3416 width=71)
2	Filter: (((scheduled_departure)::date >= '2023-08-17'::date) AND ((scheduled_departure)::date <= '2023-08-18'::date))

Figure 5-4. A plan that does not use the index due to the column transformation

Why doesn't PostgreSQL use the index? Because when the timestamp is converted to a date, a column transformation has been performed.

So is an additional functional index on `scheduled_departure::date` needed? Not necessarily. What does this selection criterion mean? It means that we want to select flights that depart on these two specific dates, no matter the time of day. This means that the flight could depart any time between midnight of August 17, 2023, and midnight of August 19, 2023. In order to make the existing index work, the selection criteria can be modified to

```
SELECT *
FROM flight
WHERE scheduled_departure >='2023-08-17'
      AND scheduled_departure <'2023-08-19'
```

Figure 5-5 shows how the execution plan has changed.

	QUERY PLAN text
1	Bitmap Heap Scan on flight (cost=121.38..9057.81 rows=8288 width=71)
2	Recheck Cond: ((scheduled_departure >= '2023-08-17 00:00:00-05':timestamp with time zone) AND (scheduled_departure < '2023-08-19 00:00:00-05':timestamp with time zone))
3	-> Bitmap Index Scan on flight_scheduled_departure (cost=0.00..119.30 rows=8288 width=0)
4	Index Cond: ((scheduled_departure >= '2023-08-17 00:00:00-05':timestamp with time zone) AND (scheduled_departure < '2023-08-19 00:00:00-05':timestamp with time zone))

Figure 5-5. An execution plan that uses the index

Looking at the execution plan, you can see that the cost estimate for a plan with index-based access is more than twice less than a sequential scan (13857.42 vs. 30474). What is more important, the execution time supports this observation: 0.5 seconds for index-based access vs. 1.5 seconds for a sequential scan.

Pay very close attention to this example. When you read about this example in a book, the preceding paragraph looks obvious. However, numerous SQL developers and report writers use similar search conditions over and over again. One frequent use case is changes made to a table today. Ninety-five percent of the time, this condition is written as `update_ts::date=CURRENT_DATE`, which successfully blocks the use of an index on the `update_ts` column. To take advantage of the index, this criterion should be written as

```
update_ts>= CURRENT_DATE
```

or, if it is possible for values of this timestamp to be in the future, the condition should be written as

```
WHERE update_ts>= CURRENT_DATE AND update_ts< CURRENT_DATE +1
```

Let’s examine another example where column transformation often remains unnoticed. Let’s say today is August 17, 2023. We are looking for flights that have departed or are scheduled to depart today. We know that for flights that have not yet departed, the `actual_departure` column may be null.

The `coalesce()` function in PostgreSQL allows us to use a different value when the first argument is null. Thus, `coalesce(actual_departure, scheduled_departure)` will return `actual_departure` if it is not null and `scheduled_departure` otherwise. Both the `scheduled_departure` and `actual_departure` columns are indexed, and you might expect these indexes to be used. For example, look at the execution plan for the following SQL statement shown in Figure 5-6:

```
SELECT *
FROM flight
WHERE coalesce(actual_departure, scheduled_departure)
      BETWEEN '2023-08-17' AND '2023-08-18'
```

	QUERY PLAN text
1	Seq Scan on flight (cost=0.00..18814.67 rows=3416 width=71)
2	Filter: ((COALESCE(actual_departure, scheduled_departure) >= '2023-08-17 00:00:00-05':timestamp with time zone) AND (COALESCE(actual_departure

Figure 5-6. A plan with a sequential scan, when indexes are present

Why aren’t any indexes utilized? Because `coalesce()` is a function, which modifies column values. Should we create another functional index? We can, but it is not really necessary. Instead, we can rewrite this SQL statement as shown in Listing 5-8, which will result in the execution plan in Figure 5-7.

Listing 5-8. Query that uses both indexes

```
SELECT * FROM flight
WHERE (actual_departure
      BETWEEN '2023-08-17' AND '2023-08-18')
      OR (actual_departure IS NULL
          AND scheduled_departure BETWEEN '2023-08-17' AND '2023-08-18')
```

	QUERY PLAN text
1	Bitmap Heap Scan on flight (cost=67.75..7271.32 rows=2395 width=71)
2	Recheck Cond: (((actual_departure >= '2023-08-17 00:00:00-05':timestamp with time zone) AND (actual_departure <= '2023-08-18 00:00:00-05':timestamp with time zone))
3	Filter: (((actual_departure >= '2023-08-17 00:00:00-05':timestamp with time zone) AND (actual_departure <= '2023-08-18 00:00:00-05':timestamp with time zone)) O
4	-> BitmapOr (cost=67.75..67.75 rows=4169 width=0)
5	-> Bitmap Index Scan on flight_actual_departure (cost=0.00..4.43 rows=1 width=0)
6	Index Cond: ((actual_departure >= '2023-08-17 00:00:00-05':timestamp with time zone) AND (actual_departure <= '2023-08-18 00:00:00-05':timestamp with ti
7	-> Bitmap Index Scan on flight_scheduled_departure (cost=0.00..62.11 rows=4169 width=0)
8	Index Cond: ((scheduled_departure >= '2023-08-17 00:00:00-05':timestamp with time zone) AND (scheduled_departure <= '2023-08-18 00:00:00-05':timestamp

Figure 5-7. Execution plan for the query from Listing 5-8

Indexes and the *like* Operator

Another group of search conditions that are not a straightforward comparison of column value constants are searches using the `like` operator. For example, the query

```
SELECT * FROM account
WHERE lower(last_name) like 'johns%';
```

will yield all accounts for which the last name begins with “johns.” In the `postgres_air` schema, the list of returned last names is

```
"Johnson"
"Johns"
"johns"
"Johnston"
"JOHNSTON"
"JOHNS"
"JOHNSON"
"johnston"
"johnson"
```

The only problem with this query is that it won’t utilize the functional index we created in the previous section, because B-tree indexes do not support searches with the “`like`” operator. Once again, if we check the execution plan for this query, we will see a sequential scan of the `account` table.

How can we solve this problem and avoid a scan?

One possible solution is to rewrite the query, replacing like with two conditions:

```
SELECT * FROM account
WHERE (lower(last_name) >='johns' and lower(last_name) < 'johnt')
```

The execution plan for this query is shown in Figure 5-8, and we can see that this plan uses an existing index.

QUERY PLAN	
	text
1	Bitmap Heap Scan on account (cost=83.20..3905.42 rows=2222 width=53)
2	Recheck Cond: ((lower(last_name) >= 'johns') AND (lower(last_name) < 'johnt'))
3	-> Bitmap Index Scan on account_last_name_lower (cost=0.00..82.64 rows=2222 width=0)
4	Index Cond: ((lower(last_name) >= 'johns') AND (lower(last_name) < 'johnt'))

Figure 5-8. The plan for a rewritten query that uses an index

A better solution would be to create a *pattern search* index:

```
CREATE INDEX account_last_name_lower_pattern
ON account (lower(last_name) text_pattern_ops);
```

Why is this index necessary? Because comparison of text values depends on the *locale*, a set of rules about character ordering, formatting, and similar things that vary by language and country. Although some may think what we have in US English is the universal order of things, it is not. The only locale that would allow us to use a B-tree index is a “C” locale, which is a standards-compliant default locale. Only strict ASCII characters are valid in this locale.

To see which locale was defined when the database was created, you need to run the command

```
SHOW LC_COLLATE;
```

And if you reside in the United States, there’s a good chance you will see

```
"en_US.UTF-8"
```

This newly created index will be utilized by queries that use the like operator. The new execution plan for our original query is shown in Figure 5-9, and we can see that it takes advantage of the new index.

	QUERY PLAN
	text
1	Bitmap Heap Scan on account (cost=83.61..3894.73 rows=3889 width=53)
2	Filter: (lower(last_name) ~~ 'johns%':text)
3	-> Bitmap Index Scan on account_last_name_lower_pattern (cost=0.00..82.64 rows=2222 width=0)
4	Index Cond: ((lower(last_name) ~>= 'johns':text) AND (lower(last_name) ~<= 'johns':text))

Figure 5-9. An execution plan with a pattern index

Using Multiple Indexes

In Figure 5-7, we see an execution plan that uses two indexes on the same table—flight. The discussion of index-based access in Chapter 3 was primarily concerned with the case of a single index. What happens when there is more than one available? How exactly does PostgreSQL use them efficiently?

The answer is in the word *bitmap*, as seen in the execution plan. Creating in-memory bitmaps allows the optimizer to use multiple indexes on one table to speed up data access. Let's look at the query with three filtering criteria for one table, all of which are supported by indexes.

Listing 5-9. A query with three filters on one table

```
SELECT
    scheduled_departure ,
    scheduled_arrival
FROM flight
WHERE departure_airport='ORD'
    AND arrival_airport='JFK'
    AND scheduled_departure BETWEEN '2023-07-03' AND '2023-07-04';
```

The execution plan for this query is shown in Figure 5-10.

	QUERY PLAN text
1	Bitmap Heap Scan on flight (cost=328.84..332.86 rows=1 width=16)
2	Recheck Cond: ((scheduled_departure >= '2023-07-03 00:00:00-05':timestamp with time zone) AND (scheduled_departure <= '2023-07-04 00:00:00-05':t
3	-> BitmapAnd (cost=328.84..328.84 rows=1 width=0)
4	-> Bitmap Index Scan on flight_scheduled_departure (cost=0.00..52.12 rows=3570 width=0)
5	Index Cond: ((scheduled_departure >= '2023-07-03 00:00:00-05':timestamp with time zone) AND (scheduled_departure <= '2023-07-04 00:00:00-05':t
6	-> Bitmap Index Scan on flight_arrival_airport (cost=0.00..125.31 rows=11318 width=0)
7	Index Cond: (arrival_airport = 'JFK':bpchar)
8	-> Bitmap Index Scan on flight_departure_airport (cost=0.00..150.91 rows=13664 width=0)
9	Index Cond: (departure_airport = 'ORD':bpchar)

Figure 5-10. Execution plan with multiple index scans on one table

As discussed in depth in Chapter 3, Postgres can use the search results from multiple indexes by creating a bitmap of blocks with matching records in main memory and then OR-ing or AND-ing them. After this process is completed, the only blocks left are the blocks that satisfy all search criteria, and PostgreSQL reads all the records in the remaining blocks to recheck the search conditions.

The blocks will be scanned in physical order, so the index-based ordering will be lost. Using a bitmap AND and OR of several index-based searches is a very efficient mechanism of applying multiple filters, but not the only one. In the next section, we will discuss another option—creating compound indexes.

Compound Indexes

So far, the indexes shown have been on individual columns. This section discusses indexes built on multiple columns and their advantages.

How Do Compound Indexes Work?

Let’s return to the query in Listing 5-9. The result of three search criteria applied to the table flight can be computed by using multiple indexes. Another option would be to create a compound index on all three columns:

```
CREATE INDEX flight_depart_arr_sched_dep ON flight(
    departure_airport,
    arrival_airport,
    scheduled_departure)
```

With this index, the execution plan would be as shown in Figure 5-11.

	QUERY PLAN text
1	Index Scan using flight_depart_arr_sched_dep on flight (cost=0.42..8.45 rows=1 width=16)
2	Index Cond: ((departure_airport = 'ORD'::bpchar) AND (arrival_airport = 'JFK'::bpchar) AND (scheduled_departure >= '2023-07-03

Figure 5-11. *A plan that uses a compound index*

This new compound index will support searches by departure_airport, by departure_airport and arrival_airport, and by departure_airport, arrival_airport, and scheduled_departure. It will not support, however, the searches by arrival_airport or scheduled_departure.

The query

```
SELECT
    departure_airport,
    scheduled_arrival,
    scheduled_departure
FROM flight
WHERE arrival_airport='JFK'
      AND scheduled_departure BETWEEN '2023-07-03' AND '2023-07-04'
```

...will produce the execution plan shown in Figure 5-12.

	QUERY PLAN text
1	Bitmap Heap Scan on flight (cost=177.71..400.06 rows=59 width=20)
2	Recheck Cond: ((scheduled_departure >= '2023-07-03 00:00:00-05'::timestamp with time zone) AND (scheduled_
3	-> BitmapAnd (cost=177.71..177.71 rows=59 width=0)
4	-> Bitmap Index Scan on flight_scheduled_departure (cost=0.00..52.12 rows=3570 width=0)
5	Index Cond: ((scheduled_departure >= '2023-07-03 00:00:00-05'::timestamp with time zone) AND (schedul
6	-> Bitmap Index Scan on flight_arrival_airport (cost=0.00..125.31 rows=11318 width=0)
7	Index Cond: (arrival_airport = 'JFK'::bpchar)

Figure 5-12. *A compound index is not used*

On the other hand, the query

```
SELECT
    scheduled_departure ,
    scheduled_arrival
FROM flight
WHERE departure_airport='ORD' AND arrival_airport='JFK'
    AND scheduled_arrival BETWEEN '2023-07-03' AND '2023-07-04';
```

...will use the compound index, although only for the first two columns, as shown in Figure 5-13.

	QUERY PLAN text
1	Bitmap Heap Scan on flight (cost=6.69..798.54 rows=1 width=16)
2	Recheck Cond: ((departure_airport = 'ORD'::bpchar) AND (arrival_airport = 'JFK'::bpchar))
3	Filter: ((scheduled_arrival >= '2023-07-03 00:00:00-05'::timestamp with time zone) AND (scheduled_arrival <= '2023-07-04 00:00:00-05'::timestamp with time zone))
4	-> Bitmap Index Scan on flight_depart_arr_sched_dep (cost=0.00..6.68 rows=226 width=0)
5	Index Cond: ((departure_airport = 'ORD'::bpchar) AND (arrival_airport = 'JFK'::bpchar))

Figure 5-13. A plan that uses the compound index for the first two columns

In general, an index on (X,Y,Z) will be used for searches on X, (X,Y), and (X,Y,Z) and even (X,Z) but not on Y alone and not on YZ. Thus, when a compound index is created, it’s not enough to decide which columns to include; their order must also be considered.

Why create compound indexes? After all, the previous section demonstrated that using several indexes together will work just fine. There are two major reasons to create this type of index: lower selectivity and additional data storage.

Lower Selectivity

Remember that the lower the selectivity is, the faster the search is, and when we are optimizing short queries, our goal is to avoid reading a large number of rows at any given point (even if we will be able to filter them out later). Sometimes, none of the individual column values are restrictive enough, and only a certain combination makes a query short.

In the example from the previous section, there are 12,922 flights with departure airport ORD and 10,530 flights that arrive at JFK. However, the number of flights that originate in ORD and land in JFK is only 184.

Using Indexes for Data Retrieval

When all the columns from a SELECT statement are included in a compound index, they may be retrieved without accessing the table. This is called the *index-only-scan* data retrieval method.

All of the execution plans in the previous section still needed to read records from the table after they were located using the index scan, because we still needed the values from columns that were not included in the index.

Let’s build one more compound index and include one more column:

```
CREATE INDEX flight_depart_arr_sched_dep_sched_arr
ON flight (departure_airport,
          arrival_airport,
          scheduled_departure,
          scheduled_arrival );
```

The execution plan of the query will instantaneously convert into an index-only scan as shown in Figure 5-14.

QUERY PLAN	
	text
1	Index Only Scan using flight_depart_arr_sched_dep_inc_sched_arr on flight (cost=0.42..8.45 rows=1 width=16)
2	Index Cond: ((departure_airport = 'ORD'::bpchar) AND (arrival_airport = 'JFK'::bpchar) AND (scheduled_departure >= '...

Figure 5-14. A plan with an index-only scan

Note that once again the search was on the first three columns of the index. If the search did not include the first column of the index, for example

```
SELECT
    departure_airport,
    scheduled_departure ,
    scheduled_arrival
```

```
FROM flight
WHERE arrival_airport='JFK'
      AND scheduled_departure BETWEEN '2023-07-03' AND '2023-07-04'
```

...the execution plan will revert to using several indexes with AND and OR, as shown in Figure 5-15.

	QUERY PLAN text
1	Bitmap Heap Scan on flight (cost=177.71..400.06 rows=59 width=20)
2	Recheck Cond: ((scheduled_departure >= '2023-07-03 00:00:00-05':timestamp with time zone) AND (scheduled_departure <= '2023-07-04 00:00:00-05':timestamp with time zone))
3	-> BitmapAnd (cost=177.71..177.71 rows=59 width=0)
4	-> Bitmap Index Scan on flight_scheduled_departure (cost=0.00..52.12 rows=3570 width=0)
5	Index Cond: ((scheduled_departure >= '2023-07-03 00:00:00-05':timestamp with time zone) AND (scheduled_departure <= '2023-07-04 00:00:00-05':timestamp with time zone))
6	-> Bitmap Index Scan on flight_arrival_airport (cost=0.00..125.31 rows=11318 width=0)
7	Index Cond: (arrival_airport = 'JFK':bpchar)

Figure 5-15. When a search does not include the first index column, a compound index is not used

Covering Indexes

Covering indexes were first introduced in PostgreSQL 11. These indexes can be viewed as a continuation of efforts to support the index-only-scan access method. A *covering index* is specifically designed to include the columns needed by a particular type of query that you run frequently.

In the previous section, the column `scheduled_arrival` was added to the index solely to avoid an extra trip to the table. It was not intended to be used as search criteria. In this case, a covering index can be used instead:

```
CREATE INDEX flight_depart_arr_sched_dep_inc_sched_arr ON flight
    (departure_airport,
     arrival_airport,
     scheduled_departure)
INCLUDE (scheduled_arrival);
```

The execution plan for the query

```
SELECT
    departure_airport,
```

```

        scheduled_departure ,
        scheduled_arrival
FROM flight
WHERE arrival_airport='JFK' AND departure_airport='ORD'
      AND scheduled_departure BETWEEN '2023-07-03' AND '2023-07-04'

```

will look like the one shown in Figure 5-16.

QUERY PLAN	
	text
1	Index Only Scan using flight_depart_arr_sched_dep_inc_sched_arr on flight (cost=0.42..8.45 rows=1 width=16)
2	Index Cond: ((departure_airport = 'ORD'::bpchar) AND (arrival_airport = 'JFK'::bpchar) AND (scheduled_departure >= '...

Figure 5-16. *A plan with an index-only scan of a covering index*

In cases like this one, there is not much difference between including an extra column in the index vs. creating a covering index. However, if more (or wider) columns need to be stored together with the indexed values, a covering index will likely be more compact.

Excessive Selection Criteria

Sometimes, when filtering logic is complex and involves attributes from multiple tables, it is necessary to provide additional, redundant filters to prompt the database engine to use specific indexes or reduce the size of join arguments. This practice is called using excessive selection criteria. The intent is to use this additional filter to preselect a small subset of records from a large table.

For some of these complex criteria, PostgreSQL is able to perform a query rewrite automatically.

For example, the filtering conditions in the query in Listing 5-10 combine values of attributes from the tables `flight` and `passenger`. In earlier versions of PostgreSQL, the engine could not start filtering prior to joining all the tables, because the `AND` is applied to the columns of different tables.

Listing 5-10. Query with conditions on two tables

```
SELECT
    last_name,
    first_name,
    seat
FROM boarding_pass bp
JOIN booking_leg bl USING (booking_leg_id)
JOIN flight f USING (flight_id)
JOIN booking b USING (booking_id)
JOIN passenger p USING (passenger_id)
WHERE
    (departure_airport='JFK'
     AND scheduled_departure BETWEEN
         '2023-07-10' AND '2023-07-11'
     AND last_name ='JOHNSON')
OR
(departure_airport='EDW'
 AND scheduled_departure BETWEEN '2023-07-13' AND '2023-07-14'
 AND last_name ='JOHNSTON')
```

However, now the optimizer can perform a complex query rewrite, as shown in Figure 5-17.

	QUERY PLAN text
1	Nested Loop (cost=187.56..61277.24 rows=11 width=15)
2	-> Nested Loop (cost=187.13..61262.37 rows=11 width=19)
3	Join Filter: (((f.departure_airport = 'JFK'::bpchar) AND (f.scheduled_departure >= '2023-07-10 00:00:00-05':timestamp with time zone) AND (f.scheduled_departure
4	-> Nested Loop (cost=186.69..55670.30 rows=2184 width=27)
5	-> Nested Loop (cost=186.25..20399.65 rows=1545 width=20)
6	-> Bitmap Heap Scan on flight f (cost=185.82..408.60 rows=59 width=16)
7	Recheck Cond: (((scheduled_departure >= '2023-07-10 00:00:00-05':timestamp with time zone) AND (scheduled_departure <= '2023-07-11 00:00:00-05':
8	-> BitmapOr (cost=185.82..185.82 rows=59 width=0)
9	-> BitmapAnd (cost=172.53..172.53 rows=56 width=0)
10	-> Bitmap Index Scan on flight_scheduled_departure (cost=0.00..52.91 rows=3649 width=0)
11	Index Cond: ((scheduled_departure >= '2023-07-10 00:00:00-05':timestamp with time zone) AND (scheduled_departure <= '2023-07-11 00:00:
12	-> Bitmap Index Scan on flight_departure_airport (cost=0.00..119.33 rows=10521 width=0)
13	Index Cond: (departure_airport = 'JFK'::bpchar)
14	-> Bitmap Index Scan on flight_depart_arr_sched_dep_inc_sched_arr (cost=0.00..13.28 rows=2 width=0)
15	Index Cond: ((departure_airport = 'EDW'::bpchar) AND (scheduled_departure >= '2023-07-13 00:00:00-05':timestamp with time zone) AND (sched
16	-> Index Scan using booking_leg_flight_id on booking_leg bl (cost=0.44..337.98 rows=85 width=12)
17	Index Cond: (flight_id = f.flight_id)
18	-> Index Scan using boarding_pass_booking_leg_id on boarding_pass bp (cost=0.44..22.60 rows=23 width=19)
19	Index Cond: (booking_leg_id = bl.booking_leg_id)
20	-> Index Scan using passenger_pkey on passenger p (cost=0.43..2.53 rows=1 width=16)
21	Index Cond: (passenger_id = bp.passenger_id)
22	Filter: ((last_name = 'JOHNSON'::text) OR (last_name = 'JOHNSTON'::text))
23	-> Index Only Scan using booking_pkey on booking b (cost=0.43..1.35 rows=1 width=8)
24	Index Cond: (booking_id = bl.booking_id)

Figure 5-17. Execution plan with conditions on two tables with query rewrite

Note the lines from 8 to 15. PostgreSQL rewrites the logical expression and selects all records from the table `flight` that may be needed for both conditions connected with OR.

In cases like this, the only thing to do is to let PostgreSQL do its job.

However, there are some queries that will run forever without human intervention. Let's look at the query in Listing 5-11. This query looks for flights that were more than one hour delayed (of which there should not be many). For all of these delayed flights, the query selects boarding passes issued after the scheduled departure.

Listing 5-11. Short query with hard-to-optimize filtering

```
SELECT
    bp.update_ts AS boarding_pass_issued,
    scheduled_departure,
    actual_departure,
    status
```

```
FROM flight f
JOIN booking_leg bl USING (flight_id)
JOIN boarding_pass bp USING (booking_leg_id)
WHERE bp.update_ts > scheduled_departure + interval '30 minutes'
      AND f.update_ts >= scheduled_departure - interval '1 hour'
```

This might seem like a contrived example, but it is modeled on production exception reports. Many companies have some sort of exception reporting in place to identify the abnormal system behavior. Crucially, by definition, the output of execution reports should be small. Exception reports, to be useful, should report on conditions that occur relatively rarely—otherwise, they would just be reports on business as usual.

The described situation certainly sounds abnormal, and there should not be many cases like this. However, the execution plan in Figure 5-18 has full scans of large tables and hash joins, even though all the appropriate indexes on all the tables involved exist.

QUERY PLAN	
text	
1	Hash Join (cost=822656.15..2204168.07 rows=2810405 width=162)
2	Hash Cond: (bp.booking_leg_id = bl.booking_leg_id)
3	Join Filter: (bp.update_ts > (f.scheduled_departure + '00:30:00'::interval))
4	-> Seq Scan on boarding_pass bp (cost=0.00..513696.84 rows=25293684 width=40)
5	-> Hash (cost=654903.74..654903.74 rows=5964513 width=99)
6	-> Hash Join (cost=32574.20..654903.74 rows=5964513 width=99)
7	Hash Cond: (bl.flight_id = f.flight_id)
8	-> Seq Scan on booking_leg bl (cost=0.00..328049.64 rows=17893564 width=32)
9	-> Hash (cost=27058.64..27058.64 rows=227725 width=71)
10	-> Seq Scan on flight f (cost=0.00..27058.64 rows=227725 width=71)
11	Filter: (update_ts >= (scheduled_departure - '01:00:00'::interval))

Figure 5-18. Suboptimal execution plan for a short query

So what went wrong?

Let’s go back to the definition of a short query. It seemed very clear in the beginning, but now it becomes a bit trickier. Recall that a query is short if it needs a small number of rows to compute results. Indeed, in this case, the number of rows we need is small, but there is no easy way to find them. So here is the caveat: it is not just that a short query

requires a small number of rows, but also that the number of rows in the result of any intermediate operation should also be small. If a query with three joins is short and, after executing the first of the joins, the intermediate result is huge, it means that something is wrong with the execution plan.

As previously discussed, the only way to read a small number of rows from a table is to use an index. However, we do not have any indexes that would support the filtering conditions in the query from Listing 5-11. Moreover, it is not possible to build such an index, because selection criteria from one table depend on values from another table. In this case, there is no way to make a selection before joining, which results in full scans and hash joins.

How can this query be improved? The answer to this question is not directly related to SQL. Chapter 1 stated that database optimization starts from gathering requirements, and this is a case where gathering precise requirements is the best path toward optimization.

Notice that, in the original query, the search space is all the flights since the dawn of time—or at least, for the entire time period captured by the database. However, this is an exception report, which most likely is reviewed on a regular cadence, and, likely, the business owner of this report is interested in recent cases since the last review. Earlier exceptions would have already appeared in previous reports and hopefully have been addressed. The next step would be to connect with the business owner of this report and ask whether a report including only the most recent exceptions suits their needs.

If the answer is yes, the excessive selection criterion we just got from business can be applied to the query. Also, we need one more index:

```
CREATE INDEX boarding_pass_update_ts ON postgres_air.boarding_
pass (update_ts);
```

Listing 5-12 shows the modified query, retaining two days of exceptions.

Listing 5-12. Query with added excessive selection criteria

```
SELECT
  bp.update_ts AS boarding_pass_issued,
  scheduled_departure,
  actual_departure,
  status
FROM flight f
```

```
JOIN booking_leg bl USING (flight_id)
JOIN boarding_pass bp USING (booking_leg_id)
WHERE bp.update_ts > scheduled_departure + interval '30 minutes'
      AND f.update_ts >=scheduled_departure -interval '1 hour';
      AND bp.update_ts >='2023-08-13' AND bp.update_ts< '2023-08-14';
```

Now, the search by timestamps will be applied first, as seen in the execution plan in Figure 5-19.

	QUERY PLAN text
1	Nested Loop (cost=664.78..321665.97 rows=3463 width=35)
2	Join Filter: (bp.update_ts > (f.scheduled_departure + '00:30:00'::interval))
3	-> Nested Loop (cost=664.36..307178.01 rows=31168 width=12)
4	-> Bitmap Heap Scan on boarding_pass bp (cost=663.91..89143.01 rows=31168 width=16)
5	Recheck Cond: ((update_ts >= '2023-08-13 00:00:00-05'::timestamp with time zone) AND (update_ts < '2023-08-14 00:00:00-05'::timestan
6	-> Bitmap Index Scan on boarding_pass_update_ts (cost=0.00..656.12 rows=31168 width=0)
7	Index Cond: ((update_ts >= '2023-08-13 00:00:00-05'::timestamp with time zone) AND (update_ts < '2023-08-14 00:00:00-05'::timestar
8	-> Memoize (cost=0.45..7.08 rows=1 width=8)
9	Cache Key: bp.booking_leg_id
10	Cache Mode: logical
11	-> Index Scan using booking_leg_pkey on booking_leg bl (cost=0.44..7.07 rows=1 width=8)
12	Index Cond: (booking_leg_id = bp.booking_leg_id)
13	-> Index Scan using flight_pkey on flight f (cost=0.42..0.45 rows=1 width=31)
14	Index Cond: (flight_id = bl.flight_id)
15	Filter: (update_ts >= (scheduled_departure - '01:00:00'::interval))

Figure 5-19. Execution plan with excessive selection criteria

The execution time for this query is less than 200 milliseconds, while the execution time for the original query was 2 minutes and 44 seconds.

A *Memoize* node (lines 8–10) will appear in the execution plan if you run PostgreSQL version 14 or higher and have a configuration parameter *enable_memoize* on. Memoized plans allow caching results from parameterized scans inside nested loop joins. This does not change the index-based access utilized in this plan.

Partial Indexes

Partial indexes are among the best features of PostgreSQL. As implied by the name, a partial index is built on a subset of a table, defined by the `WHERE` clause of the `CREATE INDEX` operator.

For example, for flights scheduled in the future, the `actual_departure` column is null. To improve search by `actual_departure`, we can create an index for only flights with a non-null actual departure value:

```
CREATE INDEX flight_actual_departure_not_null
  ON flight(actual_departure)
  WHERE actual_departure IS NOT NULL
```

In this particular case, the difference in execution time won't be dramatic, because the flight table is not very large and, in the current distribution, only half of the flights have a null actual departure. However, if the values in a column are distributed unevenly, using a partial index can provide a great advantage.

For example, the column `status` in the `flight` table has only three possible values: "On schedule," "Delayed," and "Canceled." These values are unevenly distributed; there are significantly more flights with status "On schedule" than the other two. Creating an index on the column `status` would be impractical due to the very high selectivity of this column. However, it would be nice to be able to quickly filter out the canceled flights, especially because in contrast to real life, there are not that many canceled flights in the `postgres_air` schema.

We are going to create an index:

```
CREATE INDEX flight_canceled ON flight(flight_id)
WHERE status='Canceled';
```

This index will be used in all queries where we select canceled flights, regardless of any other filtering conditions, for example:

```
SELECT * FROM flight WHERE
  scheduled_departure between '2023-08-13' AND '2023-08-14'
  AND status='Canceled'
```

The execution plan for this query is shown in [Figure 5-20](#).

	QUERY PLAN text
1	Bitmap Heap Scan on flight (cost=60.11..64.12 rows=1 width=71)
2	Recheck Cond: ((status = 'Canceled'::text) AND (scheduled_departure >= '2023-08-13 00:00:00-05'::timestamp with time zone) AND (scheduled_dep:
3	-> BitmapAnd (cost=60.11..60.11 rows=1 width=0)
4	-> Bitmap Index Scan on flight_canceled (cost=0.00..9.00 rows=182 width=0)
5	-> Bitmap Index Scan on flight_scheduled_departure (cost=0.00..50.85 rows=3443 width=0)
6	Index Cond: ((scheduled_departure >= '2023-08-13 00:00:00-05'::timestamp with time zone) AND (scheduled_departure <= '2023-08-14 00:00:

Figure 5-20. *The usage of a partial index*

Using the partial index decreases execution time from 0.72 seconds to 0.16 seconds.

Indexes and Join Order

As mentioned earlier, in short queries, the optimization goal is to avoid large intermediate results. That means ensuring that the most restrictive selection criteria are applied first. After that, for each join operation, we should ensure that the result continues to be small.

The size of join results may be small either because of restrictions on the joined tables (small number of records in join arguments) or because of a semi-join (one argument significantly restricts the result size).

Most of the time, the query planner chooses a correct join order, unless the wrong order is forced.

Let’s start with creation of several more indexes:

```
CREATE INDEX account_login ON account(login);
CREATE INDEX account_login_lower_pattern ON account (lower(login) text_
pattern_ops);
CREATE INDEX passenger_last_name ON passenger (last_name);
CREATE INDEX boarding_pass_passenger_id ON boarding_pass (passenger_id);
CREATE INDEX passenger_last_name_lower_pattern ON passenger (lower(last_
name) text_pattern_ops);
CREATE INDEX passenger_booking_id ON passenger(booking_id);
CREATE INDEX booking_account_id ON booking(account_id);
CREATE INDEX booking_email_lower_pattern ON booking
(lower(email) text_pattern_ops);
```

Now, consider the example in Listing 5-13.

Listing 5-13. Order of joins example

```

SELECT
    b.account_id,
    a.login,
    p.last_name,
    p.first_name
FROM passenger p
JOIN booking b USING(booking_id)
JOIN account a ON a.account_id=b.account_id
WHERE lower(p.last_name) LIKE 'smith%'
      AND lower(login) LIKE 'smith%';

```

The execution plan for this query is shown in Figure 5-21. Note that although the first table listed is the table `passenger` and that the first selection criterion is applied to the same table, the execution starts with the table `account`.

	QUERY PLAN text
1	Nested Loop (cost=329.77..13041.92 rows=46 width=41)
2	-> Nested Loop (cost=329.34..12588.28 rows=574 width=37)
3	-> Bitmap Heap Scan on account a (cost=328.91..9077.97 rows=88 width=29)
4	Filter: ((lower(login) ~~ 'smith% '::text) AND (lower(login) ~~ 'smith% '::text))
5	-> Bitmap Index Scan on account_login_lower_pattern (cost=0.00..328.89 rows=8564 width=0)
6	Index Cond: ((lower(login) ~>=~ 'smith% '::text) AND (lower(login) ~<~ 'smith% '::text) AND (lower(login) ~>=~ 'smith% '::text) AN
7	-> Index Scan using booking_account_id on booking b (cost=0.43..39.80 rows=9 width=12)
8	Index Cond: (account_id = a.account_id)
9	-> Index Scan using passenger_booking_id on passenger p (cost=0.43..0.78 rows=1 width=16)
10	Index Cond: (booking_id = b.booking_id)
11	Filter: (lower(last_name) ~~ 'smith% '::text)

Figure 5-21. Order of joins: execution starts from the smaller table, when selectivity is similar

The reason is that the table `account` contains significantly fewer records than the `passenger` table, and although the selectivity of both filters is approximately the same, the corresponding index on the `account` table will yield fewer records. Note that in this case the pattern index on the `passenger` table is not used at all.

However, the execution plan changes significantly when the criteria look for passengers with an uncommon last name—that is, a last name with a very low selectivity. The execution plan in Figure 5-22 indicates that in this case, starting processing from the passenger table is more restrictive. In this case, the PostgreSQL query planner uses the pattern index on the passenger table and does not use the pattern index on the account table.

	QUERY PLAN	
	text	
1	Nested Loop (cost=1.29..9127.81 rows=1 width=41)	
2	-> Nested Loop (cost=0.87..8648.91 rows=1041 width=16)	
3	-> Index Scan using passenger_last_name_lower_pattern on passenger p (cost=0.43..8.46 rows=1041 width=...	
4	Index Cond: ((lower(last_name) ~>=~ 'davey'::text) AND (lower(last_name) ~<~ 'davez'::text))	
5	Filter: (lower(last_name) ~~ 'davey%'::text)	
6	-> Index Scan using booking_pkey on booking b (cost=0.43..8.30 rows=1 width=12)	
7	Index Cond: (booking_id = p.booking_id)	
8	-> Index Scan using account_pkey on account a (cost=0.42..0.46 rows=1 width=29)	
9	Index Cond: (account_id = b.account_id)	
10	Filter: ((lower(login) ~~ 'smith%'::text) AND (lower(login) ~~ 'smith%'::text))	

Figure 5-22. *Different selectivity prompts a different join order*

Another thing worth noting while comparing these two plans is that since the first query yields significantly more rows, a bitmap index scan is used, while the second query uses the regular index scan due to the lower selectivity.

The SELECT statement in Listing 5-14 is similar, but instead of joining with the passenger table, the join is to the frequent_flyer table, which is approximately half of the size of the account table. Of course, to be able to search this table, two more indexes are required:

```
CREATE INDEX frequent_fl_last_name_lower_pattern ON frequent_flyer (lower(last_name) text_pattern_ops);

CREATE INDEX frequent_fl_last_name_lower ON frequent_flyer (lower(last_name));
```

In this case, execution will start from the table frequent_flyer, as shown in Figure 5-23.

Listing 5-14. Query selecting the number of bookings for each frequent flyer

```

SELECT
    a.account_id,
    a.login,
    f.last_name,
    f.first_name,
    count(*) AS num_bookings
FROM frequent_flyer f
JOIN account a USING(frequent_flyer_id)
JOIN booking b USING(account_id)
WHERE lower(f.last_name)='smith'
AND lower(login) LIKE 'smith%'
GROUP BY 1,2,3,4

```

	QUERY PLAN text
1	GroupAggregate (cost=4388.12..4400.94 rows=570 width=49)
2	Group Key: a.account_id, f.last_name, f.first_name
3	-> Sort (cost=4388.12..4389.54 rows=570 width=41)
4	Sort Key: a.account_id, f.last_name, f.first_name
5	-> Nested Loop (cost=1389.66..4362.03 rows=570 width=41)
6	-> Hash Join (cost=1389.23..4200.87 rows=26 width=41)
7	Hash Cond: (a.frequent_flyer_id = f.frequent_flyer_id)
8	-> Bitmap Heap Scan on account a (cost=87.45..2885.45 rows=5199 width=33)
9	Filter: (lower(login) ~~ 'smith%':text)
10	-> Bitmap Index Scan on account_login_lower_pattern (cost=0.00..86.15 rows=2573 width=0)
11	Index Cond: ((lower(login) ~>= ~ 'smith':text) AND (lower(login) ~< ~ 'smi':text))
12	-> Hash (cost=1293.75..1293.75 rows=642 width=16)
13	-> Bitmap Heap Scan on frequent_flyer f (cost=13.39..1293.75 rows=642 width=16)
14	Recheck Cond: (lower(last_name) = 'smith':text)
15	-> Bitmap Index Scan on frequent_fl_last_name_lower (cost=0.00..13.23 rows=642 width=0)
16	Index Cond: (lower(last_name) = 'smith':text)
17	-> Index Only Scan using booking_account_id on booking b (cost=0.43..5.35 rows=85 width=4)
18	Index Cond: (account_id = a.account_id)

Figure 5-23. The execution plan for the query in Listing 5-14

When Are Indexes Not Used

So far, this chapter has covered how indexes are used in queries. This section turns to situations where indexes are *not used*. Specifically, it discusses two situations: how to prevent PostgreSQL from using indexes in some circumstances and what to do when an index isn't being used and we think it ought to be.

Avoiding Index Usage

Why would it be desirable to avoid using an index? Often, database developers believe that using indexes improves the performance of any query. We can each recall situations where we were asked “to build some indexes to make this query run faster.” However, an index is not always needed and in some cases may be counterproductive. Two examples from earlier in this chapter (Figures 5-1 and 5-6) showed execution plans with sequential reads that are nevertheless quite efficient.

The two main reasons we may want to avoid using indexes are as follows:

- A small table is completely read into main memory.
- We need a large proportion of the rows of a table to execute a query.

Is there a way to avoid using existing indexes? Most of the time, the optimizer is smart enough to figure out when it should or shouldn't use indexes. But on the rare occasions when it fails, we can modify selection criteria. Recall from the beginning of this chapter that column transformations can block the usage of indexes. At that time, it was framed as a negative impact of column transformation, but it can also be used to improve performance when the goal is to block index usage.

If a column is of a numeric type, it can be modified by adding zero to its value. For example, the condition `attr1+0=p_value` will block the usage of an index on column `attr1`. For any data type, the `coalesce()` function will always block the usage of indexes, so assuming `attr2` is not nullable, the condition can be modified to something like `coalesce(t1.attr2, '0')=coalesce(t2.attr2, '0')`.

Why Does PostgreSQL Ignore My Index?

Occasionally, there are extraordinarily frustrating cases, when the appropriate index exists, but for some reason, PostgreSQL is not using it. This is the moment when a database developer with experience with other systems that allow optimizer hints

might start really missing them. However, most of the time, there is no reason for frustration. Having one of the best optimizers, PostgreSQL does the right thing in most cases. So, most likely, there is a good reason, and it is possible to find by examining the execution plan.

Let's consider an example. This example, as well as some examples in subsequent chapters, uses larger tables, which are not included in the `postgres_air` distribution due to their size. These tables are necessary to illustrate cases that occur in real life and that you might come across. Here, the table `boarding_pass_large` is being used, which has the same structure as the `boarding_pass` table, but contains three times as many rows—over 75,000,000 boarding passes. To create a larger table for experimentation, you can insert every row of the `boarding_pass` table three times.

In the `postgres_air` database, the current date is August 14, 2023. Let's select a sample of 100 passengers who checked in during the last week:

```
SELECT * FROM boarding_pass_large
WHERE update_ts::date BETWEEN '2023-08-07' AND '2023-08-14'
LIMIT 100
```

Predictably, the execution plan shown in Figure 5-24 shows a sequential scan.

	QUERY PLAN
	text 
1	Limit (cost=0.00..606.19 rows=100 width=44)
2	-> Seq Scan on boarding_pass_large (cost=0.00..2299878.44 rows=379401 width=44)
3	Filter: (((update_ts)::date >= '2023-08-07'::date) AND ((update_ts)::date <= '2023-08-14'::date))

Figure 5-24. *Sequential scan due to column transformation*

No problem, we've covered how to avoid this issue. Instead of converting the timestamp to date, we use an interval:

```
SELECT * FROM boarding_pass_large
WHERE update_ts BETWEEN '2023-08-07' AND '2023-08-15'
LIMIT 100
```

However, when we check the execution plan, we see that it still uses a sequential scan!

Why didn't removing the column transformation cause PostgreSQL to use the index? The answer is in Figure 5-25. It is the result of the combination of relatively high selectivity of this index on a large table and the presence of the LIMIT operator. The query planner estimates that the specified selection condition will select over 700,000 rows, which, recall, might require twice as many disk I/O operations. Since only 100 rows are required and since the order is not specified, it is faster to go ahead with a sequential scan of the table. There is a higher chance that the hundred records that satisfy this criterion will be found faster.

	QUERY PLAN
	text
1	Limit (cost=0.00..266.18 rows=100 width=44)
2	-> Seq Scan on boarding_pass_large (cost=0.00..1920477.08 rows=721490 width=44)
3	Filter: ((update_ts >= '2023-08-07 00:00:00-05':timestamp with time zone) AND (update_ts <= '2023-08-15 00:00:00-05':timestamp with time zone))

Figure 5-25. Sequential scan due to high index selectivity

The situation would be different if those 100 records needed to be selected in a particular order. Unless the sort order is on the indexed attribute, PostgreSQL will need to select *all* records before it can decide which ones are first.

Let's change the SELECT statement to include ordering:

```
SELECT * FROM boarding_pass_large
WHERE update_ts BETWEEN '2023-08-07' AND '2023-08-15'
ORDER BY boarding_time
LIMIT 100
```

Now the execution plan (shown in Figure 5-26) looks dramatically different.

	QUERY PLAN
	text
1	Limit (cost=1904162.11..1904162.36 rows=100 width=44)
2	-> Sort (cost=1904162.11..1905965.84 rows=721490 width=44)
3	Sort Key: boarding_time
4	-> Bitmap Heap Scan on boarding_pass_large (cost=14983.84..1876587.28 rows=721490 width=44)
5	Recheck Cond: ((update_ts >= '2023-08-07 00:00:00-05':timestamp with time zone) AND (update_ts <= '2023-08-15 00:00:00-05':timestamp with time zone))
6	-> Bitmap Index Scan on boarding_pass_large_update_ts (cost=0.00..14803.47 rows=721490 width=0)
7	Index Cond: ((update_ts >= '2023-08-07 00:00:00-05':timestamp with time zone) AND (update_ts <= '2023-08-15 00:00:00-05':timestamp with time zone))

Figure 5-26. Execution plan with sorting

Comparing the execution time for these two queries, the one with a sequential scan ran for 140 milliseconds, and the one with forced index access ran for 620 milliseconds, so the sequential scan was indeed more efficient in this case.

Let PostgreSQL Do Its Job!

In this section, we will use several modifications of the queries from the previous sections to illustrate how PostgreSQL modifies execution plans based on data statistics.

We hope that by now, a convincing case has been made that the optimizer does its job right most of the time, and our goal in most cases is just to give it enough flexibility to make the right choice.

Let's get back to the query on a large table from the previous section:

```
SELECT * FROM boarding_pass_large
WHERE update_ts BETWEEN '2023-08-07' AND '2023-08-15'
LIMIT 100
```

The execution plan for this query is shown in Figure 5-25, and the PostgreSQL optimizer chose a sequential scan, because the interval of seven days was too large to get any benefits from index access. Now, let's reduce the time interval:

```
SELECT * FROM boarding_pass_large
WHERE update_ts BETWEEN '2023-08-12' AND '2023-08-15'
LIMIT 100
```

The execution plan for this query in Figure 5-27 shows that index-based access is used.

	QUERY PLAN text
1	Limit (cost=0.57..342.92 rows=100 width=44)
2	-> Index Scan using boarding_pass_large.update_ts on boarding_pass_large (cost=0.57..798572.71 rows=233262 width=44)
3	Index Cond: ((update_ts >= '2023-08-12 00:00:00-05':timestamp with time zone) AND (update_ts <= '2023-08-15 00:00:00-05':timeste

Figure 5-27. A plan changes to index access when the time interval is smaller

Continuing to check different intervals, we will see that eight days is the pivot point in this case. If the start of the interval is any date after August 7, the execution plan will show index usage.

Even more interestingly, if `LIMIT 100` is removed from the query, the execution plan will show an index scan. If we increase the interval by just one more day, the execution plan will divert to a bitmap scan, and if we increase it for one more day, it will flip to the sequential scan (see the corresponding execution plans in Figures 5-28, 5-29, and 5-30).



The screenshot shows a database interface with a toolbar at the top containing icons for query editing, saving, and execution. Below the toolbar is a table titled "QUERY PLAN" with a "text" column. The table contains two rows:

	QUERY PLAN text
1	Index Scan using boarding_pass_large_update_ts on boarding_pass_large (cost=0.57..1745769.79 rows=623844 width=44)
2	Index Cond: ((update_ts >= '2023-08-08 00:00:00-05':timestamp with time zone) AND (update_ts <= '2023-08-15 00:00:00-05':timestamp with t

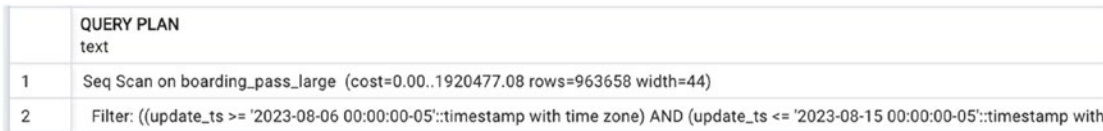
Figure 5-28. Execution plan using an index scan



The screenshot shows a database interface with a toolbar at the top. Below the toolbar is a table titled "QUERY PLAN" with a "text" column. The table contains four rows:

	QUERY PLAN text
1	Bitmap Heap Scan on boarding_pass_large (cost=14983.84..1876587.28 rows=721490 width=44)
2	Recheck Cond: ((update_ts >= '2023-08-07 00:00:00-05':timestamp with time zone) AND (update_ts <= '2023-08-15 00:00:00-05':timestamp with time z...
3	-> Bitmap Index Scan on boarding_pass_large_update_ts (cost=0.00..14803.47 rows=721490 width=0)
4	Index Cond: ((update_ts >= '2023-08-07 00:00:00-05':timestamp with time zone) AND (update_ts <= '2023-08-15 00:00:00-05':timestamp with time z...

Figure 5-29. Execution plan using a bitmap scan



The screenshot shows a database interface with a toolbar at the top. Below the toolbar is a table titled "QUERY PLAN" with a "text" column. The table contains two rows:

	QUERY PLAN text
1	Seq Scan on boarding_pass_large (cost=0.00..1920477.08 rows=963658 width=44)
2	Filter: ((update_ts >= '2023-08-06 00:00:00-05':timestamp with time zone) AND (update_ts <= '2023-08-15 00:00:00-05':timestamp with

Figure 5-30. Execution plan using a sequential scan

Let’s see another example—the query in Listing 5-13. We observed that depending on how selective the last name of the passenger is, the order of joins (and applying indexes) will change. In fact, experimenting with different last names, it is possible to identify the selectivity at which the execution plan flips (around 200 occurrences).

Finally, let’s look at one relatively simple SQL statement. Three `SELECT` statements in Listing 5-15 are identical except the filtering values for each search criterion. In the first query, departure airport has high selectivity, and passenger name has low selectivity. In the second query, both values are highly selective, and in the last query, departure airport has low selectivity. The plans, shown in Figures 5-31, 5-32, and 5-33, differ in join algorithms, the order of joins, and the indexes that are used.

Listing 5-15. SELECT with three different sets of parameters

```
--#1

SELECT
    p.last_name,
    p.first_name
FROM passenger p
JOIN boarding_pass bp USING (passenger_id)
JOIN booking_Leg bl USING (booking_leg_id)
JOIN flight USING(flight_id)
WHERE departure_airport='LAX'
    AND lower(last_name)='clark'

--#2

SELECT
    p.last_name,
    p.first_name
FROM passenger p
JOIN boarding_pass bp USING (passenger_id)
JOIN booking_Leg bl USING (booking_leg_id)
JOIN flight USING(flight_id)
WHERE departure_airport='LAX'
    AND lower(last_name)='smith'

--#3

SELECT
    p.last_name,
    p.first_name
FROM passenger p
JOIN boarding_pass bp USING (passenger_id)
JOIN booking_Leg bl USING (booking_leg_id)
JOIN flight USING(flight_id)
WHERE departure_airport='FUK'
    AND lower(last_name)='smith'
```

	QUERY PLAN
	text
1	Hash Join (cost=9678.47..165905.98 rows=198 width=12)
2	Hash Cond: (bl.flight_id = flight.flight_id)
3	-> Nested Loop (cost=134.45..156334.18 rows=10582 width=16)
4	-> Nested Loop (cost=134.02..138080.14 rows=10582 width=20)
5	-> Bitmap Heap Scan on passenger p (cost=133.45..23039.81 rows=6825 width=16)
6	Recheck Cond: (lower(last_name) = 'clark':text)
7	-> Bitmap Index Scan on passenger_last_name_lower_pattern (cost=0.00..131.75 rows=6825 width=0)
8	Index Cond: (lower(last_name) = 'clark':text)
9	-> Index Scan using boarding_pass_passenger_id on boarding_pass bp (cost=0.56..16.82 rows=4 width=16)
10	Index Cond: (passenger_id = p.passenger_id)
11	-> Index Scan using booking_leg_pkey on booking_leg bl (cost=0.44..1.73 rows=1 width=8)
12	Index Cond: (booking_leg_id = bp.booking_leg_id)
13	-> Hash (cost=9384.33..9384.33 rows=12775 width=4)
14	-> Bitmap Heap Scan on flight (cost=243.43..9384.33 rows=12775 width=4)
15	Recheck Cond: (departure_airport = 'LAX':bpchar)
16	-> Bitmap Index Scan on flight_departure_airport (cost=0.00..240.24 rows=12775 width=0)
17	Index Cond: (departure_airport = 'LAX':bpchar)

Figure 5-31. An execution plan for query #1

	QUERY PLAN
	text
1	Hash Join (cost=9678.47..165905.98 rows=198 width=12)
2	Hash Cond: (bl.flight_id = flight.flight_id)
3	-> Nested Loop (cost=134.45..156334.18 rows=10582 width=16)
4	-> Nested Loop (cost=134.02..138080.14 rows=10582 width=20)
5	-> Bitmap Heap Scan on passenger p (cost=133.45..23039.81 rows=6825 width=16)
6	Recheck Cond: (lower(last_name) = 'smith':text)
7	-> Bitmap Index Scan on passenger_last_name_lower_pattern (cost=0.00..131.75 rows=6825 width=0)
8	Index Cond: (lower(last_name) = 'smith':text)
9	-> Index Scan using boarding_pass_passenger_id on boarding_pass bp (cost=0.56..16.82 rows=4 width=16)
10	Index Cond: (passenger_id = p.passenger_id)
11	-> Index Scan using booking_leg_pkey on booking_leg bl (cost=0.44..1.73 rows=1 width=8)
12	Index Cond: (booking_leg_id = bp.booking_leg_id)
13	-> Hash (cost=9384.33..9384.33 rows=12775 width=4)
14	-> Bitmap Heap Scan on flight (cost=243.43..9384.33 rows=12775 width=4)
15	Recheck Cond: (departure_airport = 'LAX':bpchar)
16	-> Bitmap Index Scan on flight_departure_airport (cost=0.00..240.24 rows=12775 width=0)
17	Index Cond: (departure_airport = 'LAX':bpchar)

Figure 5-32. An execution plan for query #2

QUERY PLAN		
	text	
1	Nested Loop (cost=12.91..217066.10 rows=360 width=12)	
2	-> Nested Loop (cost=12.47..191823.09 rows=13958 width=8)	
3	-> Nested Loop (cost=11.91..34548.25 rows=9874 width=4)	
4	-> Bitmap Heap Scan on flight (cost=11.35..1341.38 rows=377 width=4)	
5	Recheck Cond: (departure_airport = 'FUK'::bpchar)	
6	-> Bitmap Index Scan on flight_depart_arr_sched_dep_inc_sched_arr (cost=0.00..11.25 row...	
7	Index Cond: (departure_airport = 'FUK'::bpchar)	
8	-> Index Scan using booking_leg_flight_id on booking_leg bl (cost=0.56..87.17 rows=91 width=8)	
9	Index Cond: (flight_id = flight.flight_id)	
10	-> Index Scan using boarding_pass_booking_leg_id on boarding_pass bp (cost=0.56..15.70 rows=...	
11	Index Cond: (booking_leg_id = bl.booking_leg_id)	
12	-> Index Scan using passenger_pkey on passenger p (cost=0.43..1.81 rows=1 width=16)	
13	Index Cond: (passenger_id = bp.passenger_id)	
14	Filter: (lower(last_name) = 'smith'::text)	

Figure 5-33. An execution plan for query #3

How to Build the Right Index(es)

At the start of this chapter, a minimal set of indexes was defined in the `postgres_air` schema. Almost every time we wanted to improve query performance, we suggested building yet another index. All of these indexes indeed helped improve execution time. What was never discussed was whether additional justification is needed to create a new index.

To Build or Not to Build

Twenty years ago, we were more cautious when deciding whether to add yet another index. Two major reasons against creating too many indexes are that indexes take up extra space in the database and that insert and update operations become slow when there are too many indexes to update along with the record itself. The prevailing guidance used to be to drop all indexes on a table before a bulk load and then to create them again. Some textbooks on databases still offer the same recommendation—not to drop the indexes, but to be mindful regarding the number of indexes on a table.

Since then, times have changed. With current hard- and software, the situation is different. Disk storage is cheaper, disks are faster, and, in general, fast response is more valuable than saving disk space. Twenty years ago, having a table in your database with the cumulative size of its indexes exceeding the size of the table itself was a red flag. These days, it's the norm for OLTP systems. But still, the question remains: When is enough enough?

Which Indexes Are Needed?

It is challenging to provide any general recommendations regarding which indexes are necessary. In OLTP systems, response time is usually critical, and any short query should be index-supported.

We recommend creating partial and covering indexes whenever it makes sense. Partial indexes are usually smaller than regular indexes and are more likely to fit in main memory. Covering indexes save trips to the table, thus allowing the engine to perform most processing in main memory.

Extra time needed for inserts and updates is usually less critical than fast response. However, you should always watch this time and evaluate the inventory of indexes if slowness is detected. The unique/primary key indexes and foreign keys that reference other unique/primary key fields are usual culprits of slowness, as well as triggers on insert/update. In each case, you will need to evaluate the importance of data integrity vs. the speed of updates.

There are a number of queries available online that calculate the total size of indexes on each table, and most monitoring tools will alert you about excessive growth.

Which Indexes Are Not Needed?

Even though we are usually not concerned with extra disk space needed for indexes, we do not want to create database objects that are useless. The PostgreSQL catalog view `pg_stat_all_indexes` shows the total number of index uses (scans, reads, and fetches) since the last statistics reset.

Note that some primary key indexes are *never* used for data retrieval; however, they are vital for data integrity and should not be removed.

Indexes and Short Query Scalability

In this section, we will discuss how to optimize short queries so that they will remain performant when data volumes increase.

In Chapter 1, we mentioned that optimization does not stop when a query goes to production. We should continue to monitor performance and proactively identify changes in performance dynamics.

With short queries, such performance monitoring is vital, since query behavior can change dramatically when data volumes grow, especially when the speed of growth is different for different tables.

When a query is index-supported, there is at least some assurance that it is scalable, because the number of accesses to index grows only logarithmically relative to table growth. But if the size of a table grows fast, the index might grow large enough not to fit in main memory, or it might be pushed out by indexes for competing queries. If this happens, execution time may increase sharply.

It is possible that in the beginning a query works fast without any indexes, and we might not know for sure which indexes will be needed in the future. It is also possible that a condition for a partial index was very restrictive and index access was very fast, but later, with more and more records satisfying the condition, the index became less efficient.

In short, although we strive to make sure short queries are scalable and will perform well even when data volumes grow, we can't assume that anything is optimized "forever." We should always keep an eye on data volume, value distributions, and other characteristics that can interfere with performance.

Summary

This chapter covered short queries and what techniques can be used to optimize them. The primary optimization goal for short queries is to apply the most restrictive search criteria first and to ensure that all intermediate results remain small. As such, the chapter discussed the role of indexes for short queries and showed how to determine what indexes are needed to create to support specific queries.

This chapter also showed various execution plans and how to read them to understand the order of joins and filtering, as well as discussing various types of indexes available in PostgreSQL and when they can be useful. More complex index types will be considered in depth in Chapter 14.

CHAPTER 6

Long Queries and Full Scans

Chapter 5 discussed short queries and explained how to identify short queries and optimization strategies and techniques applicable to them. It also covered the importance of indexes for short queries, the most commonly used index types, and their applications. Chapter 5 also provided an opportunity to practice reading execution plans and, hopefully, acquire confidence in the PostgreSQL optimizer.

This chapter concerns long queries. Some queries just can't run in a fraction of a second, no matter how well-written. This does not mean they cannot be optimized. Many practitioners hold that since analytical reports do not have strict response time requirements, it is not important how fast or slow they run. In extreme cases, report developers make no effort to make sure that reports are complete in a reasonable time, giving the excuse that the query only runs once a day or once a week or once a month.

This is a dangerous practice. If report performance is neglected, performance can easily degrade from minutes to hours or more. We have observed reports that run for six days before completion! And by the time the situation becomes that dire, it is not trivial to fix in a limited time frame. Often, when an analytical report is developed, source data volumes are really small, and everything performs well. It is the job of SQL developers to examine execution plans even if queries are running fine now and to be proactive to prevent future performance degradation.

Which Queries Are Considered Long?

Chapter 5 introduced the formal definition of a short query. It is logical to assume that all queries that are not short are long. This is true, but a definition based on negation might not be intuitive to apply in practice.

The two examples of long queries from Chapter 5 (Listings 5-1 and 5-3) are copied here in Listings 6-1 and 6-2, respectively. The first of the two queries is a long query with massive output; the query returns every possible combination of arrival and departure airports. The second one produces only one line of output—showing the average length of flights and number of passengers for all flights in the `postgres_air` schema—but is still classified as a long query.

Listing 6-1. Long query with a large result set

```
SELECT
    d.airport_code AS departure_airport
    a.airport_code AS arrival_airport
FROM  airport a,
      airport d
WHERE a.airport_code <> d.airport_code
```

Listing 6-2. 3Long query with a one-row result set

```
SELECT
    avg(flight_length),
    avg (passengers)
FROM
    (SELECT
        flight_no,
        scheduled_arrival -scheduled_departure AS flight_length,
        count(passenger_id) passengers
    FROM flight f
        JOIN booking_leg bl ON bl.flight_id = f.flight_id
        JOIN passenger p ON p.booking_id=bl.booking_id
    GROUP BY 1,2) a
```

So what is a long query, anyway?

A query is considered *long* when query selectivity is high for at least one large table; that is, almost every row contributes to the output, even when the output size is small.

What kind of optimization goals should we have for long queries? A common misconception explicitly refuted in this chapter is that if a query is long, there is no way to significantly improve its performance. Moreover, each co-author has had the joy of improving the performance of a long query by an order of *several hundred times*. Such improvements are made possible when two optimization strategies are applied:

1. Avoid multiple table scans.
2. Reduce the size of the result at the earliest possible stage.

The remainder of this chapter explains these techniques in detail and will describe several methods to achieve this goal.

Long Queries and Full Scans

Chapter 5 stated that short queries require the presence of indexes on columns included in search criteria. For long queries, it is the converse: indexes are not needed, and if tables are indexed, we want to ensure that indexes are not used.

Why are full table scans desirable for long queries? As shown in Figure 3-3, when the number of rows needed is large enough, index access will require more I/O operations. What percentage or number of records is “large enough” varies and depends on many different factors. By now, it should not be a surprise that most of the time, PostgreSQL estimates this percentage correctly.

Chapter 5 said something very similar about short queries. But “how large is large enough” is more difficult to estimate than “how small is small enough.”

The estimate of how many records is too many evolves as better hardware, faster disks, and more powerful CPUs become available. For this reason, this book tries to avoid giving specific number thresholds that will necessarily change. To build representative cases for this chapter, several tables were built with hundreds of millions of rows of data. These are too large to be included with the `postgres_air` distribution. However, it would be unsurprising if some of the examples in this chapter are no longer representative in a few years.

Long Queries and Hash Joins

In a majority of examples in this chapter, a hash join algorithm is used. This is exactly what we hope to see in the execution plan of a long query. Why is a hash join preferable in this case? In Chapter 3, we estimated the costs of both nested loop and hash join algorithms.

For nested loops, the cost of the join of tables R and S is

$$\text{cost}(\text{nl}, R, S) = \text{size}(R) * \text{size}(S) + \text{size}(R) * \text{size}(S) / \text{size}(JA)$$

For a hash join, it is

$$\text{cost}(\text{hash}, R, S) = \text{size}(R) + \text{size}(S) + \text{size}(R) * \text{size}(S) / \text{size}(JA)$$

where JA represents the number of distinct values of the join attribute. As mentioned in Chapter 3, a third term, which represents the size of the result set, should be added to the cost of both algorithms, but for the nested loop algorithm, this value is significantly less than the cost of the join itself. For long queries, the size of R and S is large (because they are not significantly restricted), making the cost of nested loops significantly higher than the cost of a hash join.

Suppose database statistics estimate the size of table R to be 1,000,000 rows and the size of table S 2,000,000 rows and the join attribute JA has 100,000 distinct values. According to the formulas in Chapter 3, the size of the join result will be

$$Z = \text{size}(R) * \text{size}(S) / \text{size}(JA) = 20,000,000$$

Then, the number of operations of the nested loop algorithm will be

$$Z + \text{size}(R) * \text{size}(S) = 2,000,020,000,000$$

and the number of operations needed for the hash join algorithm will be

$$Z + \text{size}(R) + \text{size}(S) = 23,000,000$$

which is almost 10,000 times smaller. The planner will estimate the cost of these algorithms based on similar calculations and, of course, will choose the hash join algorithm.

Hash joins work best when the first argument fits into main memory. The size of memory available can be tuned with server parameters.

In some cases, a merge join algorithm is used, for example, in Figure 6-10 later in this chapter. Chapter 3 noted that a merge join can be more efficient when at least one of the

tables is presorted. In this case, since unique values are being selected, sorting is indeed performed.

Summarizing Chapter 5 and this chapter, most of the time, index access works well with the nested loop algorithm (and vice versa), and sequential scans work well with a hash join.

Since PostgreSQL does not have optimizer hints, is there any way to force a specific join algorithm? As already mentioned multiple times, the best thing we can do is not to restrict the optimizer in the manner we write SQL statements.

Long Queries and the Order of Joins

Join order for small queries was discussed in Chapter 5. For short queries, the desired join order is the one that would prompt the usage of indexes with lower selectivity first.

Since we do not expect indexes to be used in long queries, does the order of joins make a difference? Perhaps surprisingly, it does. Large tables can differ in size significantly. Also, in practice, when selecting “almost all records,” the word “almost” can mean as little as 30% and as much as 100%. Even when indexes are not used, the order of joins matters, because it is important to keep interim datasets as small as possible.

The most restrictive joins (i.e., joins that reduce the result set size the most) should be executed first.

The optimizer will most often choose the correct order; however, it is the responsibility of the developer to verify that the optimizer has chosen correctly.

What Is a Semi-join?

Often, the most restrictive join in a query is a *semi-join*. Let’s pause to offer a formal definition.

A semi-join between two tables R and S returns rows from table R for which there is at least one row from table S with matching values in the joining columns.

To clarify, a semi-join is not an extra SQL operation; it is not valid SQL to write a statement like `SELECT a.* FROM a SEMI JOIN b`. A semi-join is a special kind of join that satisfies two specific conditions: First, only columns from the first table appear in the result set. Second, rows from the first table are not duplicated where there is more than one match in the second table. Most often, a semi-join doesn't include the `JOIN` keyword at all. The first and most common way of specifying a semi-join is shown in Listing 6-3. This query finds all flight information for flights with at least one booking.

Listing 6-3. Defining a semi-join using the `EXISTS` keyword

```
SELECT * FROM flight f WHERE EXISTS
  (SELECT flight_id FROM booking_leg WHERE flight_id=f.flight_id)
```

This query uses an implicit join with table `booking_leg` to filter records from the table `flight`. In other words, instead of supplying values for filtering, we use column values from another table.

An equivalent query showing another way to specify a semi-join is shown in Listing 6-4.

Listing 6-4. Defining a semi-join using the `IN` keyword

```
SELECT * FROM flight WHERE flight_id IN
  (SELECT flight_id FROM booking_leg)
```

How can these queries contain joins when neither uses the `JOIN` keyword? The answer is in the execution plan, which is identical for both queries, shown in Figure 6-1.

QUERY PLAN	
	text
1	Nested Loop Semi Join (cost=0.56..438436.84 rows=197629 width=71)
2	-> Seq Scan on flight f (cost=0.00..23642.76 rows=683176 width=71)
3	-> Index Only Scan using booking_leg_flight_id on booking_leg (cost=0.56..3.32 rows=91 width=4)
4	Index Cond: (flight_id = f.flight_id)

Figure 6-1. Execution plan for a semi-join

You can see a `SEMI JOIN` in this plan, even though the keyword `JOIN` was not used in the query itself.

Although these two ways of writing queries with semi-joins are semantically identical, in PostgreSQL, only the first one guarantees the presence of SEMI JOIN in the execution plan. The plans are identical for both queries in Listings 6-3 and 6-4, but in other cases, the optimizer may choose to rewrite it as a regular join. This decision is based both on the cardinality of the relationship between two tables and filter selectivity.

Semi-joins and Join Order

Since a semi-join may significantly reduce the size of the result set and, by definition, will never increase its size, semi-joins are often the most restrictive join in the query, and as stated earlier, the most restrictive join should be executed first.

Semi-joins never increase the size of the result set; check whether it is beneficial to apply them first.

Of course, this is possible only when a semi-join condition applies to the columns of a single table. In cases when a semi-join condition references more than one table, those tables must be joined before the semi-join is applied.

Consider the example in Listing 6-5, which shows bookings departing from airports located in the United States.

Listing 6-5. Order of joins when a semi-join is present

```
SELECT departure_airport,
       booking_id,
       is_returning
FROM booking_leg bl
     JOIN flight f USING (flight_id)
WHERE departure_airport
      IN (SELECT
          airport_code
        FROM airport
        WHERE iso_country='US')
```

Figure 6-2 shows the execution plan for this query.

	QUERY PLAN	
	text	
1	Hash Join (cost=18963.54..433030.38 rows=3645943 width=9)	
2	Hash Cond: (bl.flight_id = f.flight_id)	
3	-> Seq Scan on booking_leg bl (cost=0.00..310506.66 rows=17893566 ...	
4	-> Hash (cost=17223.52..17223.52 rows=139202 width=8)	
5	-> Hash Join (cost=20.41..17223.52 rows=139202 width=8)	
6	Hash Cond: (f.departure_airport = airport.airport_code)	
7	-> Seq Scan on flight f (cost=0.00..15398.78 rows=683178 width...	
8	-> Hash (cost=18.65..18.65 rows=141 width=4)	
9	-> Seq Scan on airport (cost=0.00..18.65 rows=141 width=4)	
10	Filter: (iso_country = 'US'::text)	

Figure 6-2. Execution plan for the query from Listing 6-5

This execution plan does not show a semi-join operation, but rather a hash join, since there are no duplicates to remove. However, it’s still a logical semi-join and most restrictive, so it is executed first. It is also worth taking a brief diversion to note the sequential scan on the airport table. The sequential scan is used because there is no index on the iso_country field. Let’s create this index and see whether it will speed things up.

```
If this index exists

CREATE INDEX airport_iso_country
ON airport(iso_country);
```

...the query planner will use it, as shown in Figure 6-3. However, the execution time in this case will be the same or worse than the time with a sequential scan, because the index is not selective enough. Let’s drop this index for now.

	QUERY PLAN	
	text	
1	Hash Join (cost=18961.90..433028.74 rows=3645943 width=9)	
2	Hash Cond: (bl.flight_id = f.flight_id)	
3	-> Seq Scan on booking_leg bl (cost=0.00..310506.66 rows=17893566 width=9)	
4	-> Hash (cost=17221.87..17221.87 rows=139202 width=8)	
5	-> Hash Join (cost=18.77..17221.87 rows=139202 width=8)	
6	Hash Cond: (f.departure_airport = airport.airport_code)	
7	-> Seq Scan on flight f (cost=0.00..15398.78 rows=683178 width=8)	
8	-> Hash (cost=17.01..17.01 rows=141 width=4)	
9	-> Bitmap Heap Scan on airport (cost=5.24..17.01 rows=141 width=4)	
10	Recheck Cond: (iso_country = 'US'::text)	
11	-> Bitmap Index Scan on airport_iso_country (cost=0.00..5.21 rows=141 width=...)	
12	Index Cond: (iso_country = 'US'::text)	

Figure 6-3. Execution plan with an index scan

More on Join Order

Let's take a look at a more complex example in Listing 6-6, of a long query with more than one semi-join. This query, like the previous, finds bookings for flights departing from the United States, but is limited to bookings updated since July 1, 2020. Since we do not have an index on the `update_ts` column of the booking table, let's create it now and see whether it will be used:

```
CREATE INDEX booking_update_ts ON booking (update_ts);
```

Listing 6-6. Two semi-joins in one long query

```
SELECT
    departure_airport,
    booking_id, '
    is_returning
FROM booking_leg bl
JOIN flight f USING (flight_id)
```



```
WHERE departure_airport IN
      (SELECT airport_code
        FROM airport
        WHERE iso_country='US')
AND bl.booking_id IN
      (SELECT booking_id FROM booking
        WHERE update_ts>'2023-07-01')
```

The execution plan in Figure 6-4 shows that a semi-join on airport.iso_country is executed first. Just as in the preceding code, although we use the keyword IN, the optimizer uses JOIN, not SEMI JOIN, because there is no need to eliminate duplicates.

	QUERY PLAN
	text
1	Hash Join (cost=208078.19..674465.64 rows=1181514 width=9)
2	Hash Cond: (bl.booking_id = booking.booking_id)
3	-> Hash Join (cost=18963.54..433030.38 rows=3645943 width=9)
4	Hash Cond: (bl.flight_id = f.flight_id)
5	-> Seq Scan on booking_leg bl (cost=0.00..310506.66 rows=17893566 width=9)
6	-> Hash (cost=17223.52..17223.52 rows=139202 width=8)
7	-> Hash Join (cost=20.41..17223.52 rows=139202 width=8)
8	Hash Cond: (f.departure_airport = airport.airport_code)
9	-> Seq Scan on flight f (cost=0.00..15398.78 rows=683178 width=8)
10	-> Hash (cost=18.65..18.65 rows=141 width=4)
11	-> Seq Scan on airport (cost=0.00..18.65 rows=141 width=4)
12	Filter: (iso_country = 'US'::text)
13	-> Hash (cost=159111.20..159111.20 rows=1828756 width=8)
14	-> Seq Scan on booking (cost=0.00..159111.20 rows=1828756 width=8)
15	Filter: (update_ts > '2023-07-01 00:00:00-05'::timestamp with time zone)

Figure 6-4. Execution plan with two semi-joins

Three things in this execution plan are worth noting. First, although index-based access is used to obtain some interim results and we can see that the nested loop join algorithm is used in this case, the final join is hash based, because a significant portion of both datasets is used. Second, the semi-join uses a table sequential scan. And even

though this way we are reading all the rows from the `airport` table, the result set size is smaller than it would be if we would join flights with booking legs and filter by the airport location afterward. That's the benefit of the optimizer choosing the most restrictive semi-join.

Lastly, although there is an index on the `update_ts` column of the booking table, this index is not used, because the condition `update_ts > '2023-07-01'` covers almost half the rows in this table.

However, if we change the filtering criteria in this query (shown in Listing 6-6) and reduce the interval to `update_ts > '2023-08-10'`, the execution plan will change drastically—see Figure 6-5. In this new execution plan, we can see that not only is the filter on `update_ts` more restrictive but also the optimizer judges that it may be beneficial to use the index access.

	QUERY PLAN	
	text	
1	Hash Join (cost=102683.56..461173.60 rows=35654 width=9)	
2	Hash Cond: (bl.flight_id = f.flight_id)	
3	-> Hash Join (cost=83720.02..441197.33 rows=174985 width=9)	
4	Hash Cond: (bl.booking_id = booking.booking_id)	
5	-> Seq Scan on booking_leg bl (cost=0.00..310506.66 rows=17893566 width=9)	
6	-> Hash (cost=83030.18..83030.18 rows=55187 width=8)	
7	-> Bitmap Heap Scan on booking (cost=1036.13..83030.18 rows=55187 width=8)	
8	Recheck Cond: (update_ts > '2023-08-10 00:00:00-05':timestamp with time zone)	
9	-> Bitmap Index Scan on booking_update_ts (cost=0.00..1022.33 rows=55187 width=8)	
10	Index Cond: (update_ts > '2023-08-10 00:00:00-05':timestamp with time zone)	
11	-> Hash (cost=17223.52..17223.52 rows=139202 width=8)	
12	-> Hash Join (cost=20.41..17223.52 rows=139202 width=8)	
13	Hash Cond: (f.departure_airport = airport.airport_code)	
14	-> Seq Scan on flight f (cost=0.00..15398.78 rows=683178 width=8)	
15	-> Hash (cost=18.65..18.65 rows=141 width=4)	

Figure 6-5. Execution plan with two semi-joins with a different selectivity

Is index access to the booking table indeed the best option in this case? We can compare by blocking index access applying a column transformation to the `update_ts` column, rewriting the filter the following way: `coalesce(update_ts, '2023-08-11') > '2023-08-10'`.

As seen in Figure 6-6, this forces a sequential scan. And, in fact, blocking the index and forcing the sequential scan performs better than index access on larger time intervals. As the time interval is reduced further, index access has the advantage. ‘20230807’ appears to be a tipping point; for all dates starting from ‘2023-08-08’, the index access will work better.

	QUERY PLAN
	text
1	Hash Join (cost=208938.50..675530.96 rows=1215314 width=9)
2	Hash Cond: (bl.booking_id = booking.booking_id)
3	-> Hash Join (cost=18963.54..433030.38 rows=3645943 width=9)
4	Hash Cond: (bl.flight_id = f.flight_id)
5	-> Seq Scan on booking_leg bl (cost=0.00..310506.66 rows=17893566 width=9)
6	-> Hash (cost=17223.52..17223.52 rows=139202 width=8)
7	-> Hash Join (cost=20.41..17223.52 rows=139202 width=8)
8	Hash Cond: (f.departure_airport = airport.airport_code)
9	-> Seq Scan on flight f (cost=0.00..15398.78 rows=683178 width=8)
10	-> Hash (cost=18.65..18.65 rows=141 width=4)
11	-> Seq Scan on airport (cost=0.00..18.65 rows=141 width=4)
12	Filter: (iso_country = 'US')::text)
13	-> Hash (cost=159112.23..159112.23 rows=1881099 width=8)
14	-> Seq Scan on booking (cost=0.00..159112.23 rows=1881099 width=8)
15	Filter: (COALESCE(update_ts,'2023-08-11 00:00:00-05':timestamp with time zone) > '2023-08-10 00:00:00-05':timestamp with time z...

Figure 6-6. Forcing a full scan

What Is an Anti-join?

Just like it sounds, an ANTI JOIN is the opposite of a SEMI JOIN. Formally

An anti-join between two tables R and S returns rows from table R for which there are no rows from table S with a matching value in the joining column.

As in the case of a semi-join, there is no ANTI JOIN operator. Instead, a query with an anti-join can be written in two different ways, shown in Listings 6-7 and 6-8. These queries return flights that have no bookings.

Listing 6-7. Defining an anti-join using the NOT EXISTS keyword

```
SELECT * FROM flight f WHERE NOT EXISTS
  (SELECT flight_id FROM booking_leg WHERE flight_id=f.flight_id)
```

Listing 6-8. Defining an anti-join using NOT IN

```
SELECT * FROM flight WHERE flight_id NOT IN
  (SELECT flight_id FROM booking_leg)
```

Just as with semi-joins, although both ways of writing a query with an anti-join are semantically equivalent, in PostgreSQL, only the NOT EXISTS version guarantees the anti-join in the execution plan. Figures 6-7 and 6-8 show the execution plan for Listings 6-7 and 6-8, respectively. In this particular case, both queries will be executed in approximately the same time, and the plan with an anti-join is only slightly faster. There are no generic guidelines for which syntax for an anti-join is better. Developers should try both ways to see which will perform better in their use case.

Data Output	Explain	Messages	Notifications
	QUERY PLAN text		
1	Nested Loop Anti Join (cost=0.56..429448.81 rows=543600 width=71)		
2	-> Seq Scan on flight f (cost=0.00..23642.76 rows=683176 width=71)		
3	-> Index Only Scan using booking_leg_flight_id on booking_leg (cost=0.56..3.48 rows=128 width=4)		
4	Index Cond: (flight_id = f.flight_id)		

Figure 6-7. Execution plan with an anti-join

Data Output	Explain	Messages	Notifications
	QUERY PLAN text		
1	Seq Scan on flight (cost=0.00..232232347044.02 rows=341588 width=71)		
2	Filter: (NOT (SubPlan 1))		
3	SubPlan 1		
4	-> Materialize (cost=0.00..635290.62 rows=17828108 width=4)		
5	-> Seq Scan on booking_leg (cost=0.00..476508.08 rows=17828108 width=4)		

Figure 6-8. Execution plan without an anti-join

Semi- and Anti-joins Using the JOIN Operator

At this point, an astute reader might wonder why we can't use an explicit join and specify exactly what we need. Why use the EXISTS and IN operators? The answer is it's possible and, in some cases, it might indeed be a better solution than using semi-joins. But it takes care to construct a logically equivalent query.

The queries in Listings 6-3 and 6-4 are semantically equivalent, but Listing 6-9 is not. Recall that Listings 6-3 and 6-4 return information for flights that have at least one booking.

Listing 6-9. Join returning duplicates

```
SELECT f.*
FROM flight f
JOIN booking_leg bl USING (flight_id)
```

By contrast, Listing 6-9 will return as many rows for each flight as the number of bookings with the corresponding flight_id. To return only one record per flight, like the original query, it would need to be rewritten as shown in Listing 6-10.

Listing 6-10. Query with a join returning one row per flight

```
SELECT *
FROM flight f
JOIN (select distinct flight_id FROM booking_leg) bl USING (flight_id)
```

The execution plan for this query is shown in Figure 6-9, and it does not contain a semi-join.

QUERY PLAN	
	text
1	Hash Join (cost=40189.02..828769.06 rows=139576 width=71)
2	Hash Cond: (booking_leg.flight_id = f.flight_id)
3	-> Unique (cost=0.56..777720.45 rows=139576 width=4)
4	-> Index Only Scan using booking_leg_flight_id on booking_leg (cost=0.56..733150.18 rows=17828108 width=4)
5	-> Hash (cost=23642.76..23642.76 rows=683176 width=71)
6	-> Seq Scan on flight f (cost=0.00..23642.76 rows=683176 width=71)

Figure 6-9. Execution plan for the query in Listing 6-10

It is not obvious from the execution plan whether this query will be faster or slower than the query with a semi-join. In practice, it runs more than twice as fast as the query from Listing 6-3.

If you only need the IDs of the flights that have a booking, it may be enough to run the query in Listing 6-11.

Listing 6-11. Query with a join returning only `flight_id` with one row per flight

```
SELECT flight_id
FROM flight f
JOIN (select distinct flight_id FROM booking_leg) bl USING (flight_id)
```

The execution plan for this query, shown in Figure 6-10, differs significantly from the one in Figure 6-9, and the execution is even faster.

	QUERY PLAN
	text
1	Merge Join (cost=0.99..819305.18 rows=139576 width=4)
2	Merge Cond: (f.flight_id = booking_leg.flight_id)
3	-> Index Only Scan using flight_pkey on flight f (cost=0.42..36736.33 rows=683176 width=4)
4	-> Unique (cost=0.56..777720.45 rows=139576 width=4)
5	-> Index Only Scan using booking_leg_flight_id on booking_leg (cost=0.56..733150.18 rows=17828108 width=4)

Figure 6-10. Execution plan with a merge join

What about anti-joins? An anti-join cannot create duplicates, which means that an OUTER JOIN with subsequent filtering of NULL values can be used. Thus, the query in Listing 6-7 is equivalent to the query in Listing 6-12.

Listing 6-12. Outer join with filtering of NULL values

```
SELECT f.flight_id
FROM flight f
LEFT OUTER JOIN booking_leg bl USING (flight_id)
WHERE bl.flight_id IS NULL
```

The execution plan for this query includes an anti-join—see Figure 6-11.

QUERY PLAN	
	text
1	Nested Loop Anti Join (cost=0.56..429448.81 rows=543600 width=4)
2	-> Seq Scan on flight f (cost=0.00..23642.76 rows=683176 width=4)
3	-> Index Only Scan using booking_leg_flight_id on booking_leg bl (cost=0.56..3.48 rows=128 width=4)
4	Index Cond: (flight_id = f.flight_id)

Figure 6-11. Execution plan for the query in Listing 6-12

The optimizer recognizes this construct and rewrites it to an anti-join. This optimizer behavior is stable and can be relied upon.

When Is It Necessary to Specify Join Order?

So far, the optimizer has chosen the best join order without any intervention from the SQL developer, but this isn’t always the case.

Long queries are more likely in OLAP systems. In other words, a long query is most likely an analytical report that most likely joins a number of tables. This number, as anyone who has worked with OLAP systems can attest, can be massive. When the number of tables involved in a query becomes too large, the optimizer no longer attempts to find the best possible join order. Although most system parameters are out of the scope of this book, there is one worth mentioning: `join_collapse_limit`.

This parameter caps the number of tables in a join that will be still processed by the cost-based optimizer. The default value of this parameter is 8. This means that if the number of tables in a join is eight or fewer, the optimizer will perform a selection of candidate plans, compare plans, and choose the best one. But if the number of tables is nine or more, it will simply execute the joins in the order the tables are listed in the `SELECT` statement.

Why not set this parameter to the highest possible value? There is no official upper limit to this parameter, so it can be the maximum integer, which is 2147483647. However, the higher you set this parameter, the more time will be spent choosing the best plan. The number of possible plans to consider for a query joining n tables is $n!$ Thus, when n is 8, a maximum of 40,000 plans can be compared. If n is increased to 10, the number of plans to consider will increase to three million, and the number rises predictably from

there—when this parameter is set to 20, the total number of plans is already too big to fit the integer. One of us once observed a data scientist locally changing this parameter to 30, to deal with a query with 30 joins. The results were excruciating—not only did the execution stall but even the EXPLAIN command couldn’t return a result.

This is easy to experiment with; this parameter can be set locally on a session level, so run the command

```
SET join_collapse_limit = 10
```

and check the runtime of the EXPLAIN command.

In addition, recall that table statistics are not available for intermediate results, which may cause the optimizer to choose a suboptimal join order. If the SQL developer knows a better order of joins, it is possible to force the desired join order by setting `join_collapse_limit` to 1. In this case, the optimizer will generate a plan in which the joins will be executed in the order they appear in the SELECT statement.

Force a specific join order by setting the `join_collapse_limit` parameter to 1.

For example, if the command in Listing 6-13 is executed (i.e., an EXPLAIN on the query in Listing 6-6), the execution plan in Figure 6-12 shows that joins are executed exactly in the order they are listed, and the index on `update_ts` is not used (which in this case affects performance negatively).

Listing 6-13. Partially disabling cost-based optimization

```
SET join_collapse_limit=1;
EXPLAIN
SELECT
    departure_airport,
    booking_id,
    is_returning
FROM booking_leg bl
JOIN flight f USING (flight_id)
WHERE departure_airport IN (
    SELECT
        airport_code
    FROM airport
```



```
WHERE iso_country='US')
AND bl.booking_id IN (
    SELECT
        booking_id
    FROM booking
    WHERE update_ts>'2023-08-08')
```

	QUERY PLAN	
	text	
1	Hash Join (cost=164521.00..756238.45 rows=86548 width=9)	
2	Hash Cond: (bl.booking_id = booking.booking_id)	
3	-> Hash Join (cost=26627.92..608774.76 rows=3645943 width=9)	
4	Hash Cond: (f.departure_airport = airport.airport_code)	
5	-> Hash Join (cost=26607.50..561496.06 rows=17893566 width=9)	
6	Hash Cond: (bl.flight_id = f.flight_id)	
7	-> Seq Scan on booking_leg bl (cost=0.00..310506.66 rows=17893566 width=9)	
8	-> Hash (cost=15398.78..15398.78 rows=683178 width=8)	
9	-> Seq Scan on flight f (cost=0.00..15398.78 rows=683178 width=8)	
10	-> Hash (cost=18.65..18.65 rows=141 width=4)	
11	-> Seq Scan on airport (cost=0.00..18.65 rows=141 width=4)	
12	Filter: (iso_country = 'US'::text)	
13	-> Hash (cost=136218.57..136218.57 rows=133961 width=8)	
14	-> Bitmap Heap Scan on booking (cost=2510.63..136218.57 rows=133961 width=8)	
15	Recheck Cond: (update_ts > '2023-08-08 00:00:00-05'::timestamp with time zone)	
16	-> Bitmap Index Scan on booking_update_ts (cost=0.00..2477.14 rows=133961 width=...)	
17	Index Cond: (update_ts > '2023-08-08 00:00:00-05'::timestamp with time zone)	

Figure 6-12. Execution plan with disabled cost-based optimization

Another way to force a specific join order is using *common table expressions*, which will be discussed in Chapter 7.

Grouping: Filter First, Group Last

In Chapter 5, we mentioned that for short queries, grouping is not time-consuming. For long queries, the way we approach grouping may have a very significant impact on performance. Suboptimal decisions regarding the point at which grouping is performed often become a major source of overall query slowness.

Listing 6-14 shows a query that calculates the average price of every trip and the total number of passengers for each flight for all flights with any bookings.

Listing 6-14. Average ticket price and total number of passengers per flight

```
SELECT
    bl.flight_id,
    departure_airport,
    (avg(price))::numeric (7,2) AS avg_price,
    count(DISTINCT passenger_id) AS num_passengers
FROM booking b
JOIN booking_leg bl USING (booking_id)
JOIN flight f USING (flight_id)
JOIN passenger p USING (booking_id)
GROUP BY 1,2
```

To calculate these numbers for just one flight, a common anti-pattern is the query in Listing 6-15.

Listing 6-15. Average ticket price and total number of passengers on a specific flight

```
SELECT *
FROM
    (SELECT
        bl.flight_id,
        departure_airport,
        (avg(price))::numeric (7,2) AS avg_price,
        count(DISTINCT passenger_id) AS num_passengers
    FROM booking b
    JOIN booking_leg bl USING (booking_id)
```

```

        JOIN flight f USING (flight_id)
        JOIN passenger p USING (booking_id)
        GROUP BY 1,2) a
WHERE flight_id=222183

```

In this query, we select the data for one flight from an inline SELECT. Earlier versions of PostgreSQL could not process such constructs efficiently. The database engine would first execute the inner SELECT with grouping and only then select the line that corresponds to the specific flight. To make sure the query is executed efficiently, one would need to write it as shown in Listing 6-16.

Listing 6-16. Pushing a condition inside the GROUP BY

```

SELECT
    bl.flight_id,
    departure_airport,
    (avg(price))::numeric (7,2) AS avg_price,
    count(DISTINCT passenger_id) AS num_passengers
FROM booking b
JOIN booking_leg bl USING (booking_id)
JOIN flight f USING (flight_id)
JOIN passenger p USING (booking_id)
WHERE flight_id=222183
GROUP BY 1,2

```

But now, due to ongoing improvements to the optimizer, both queries will be executed using the execution plan in Figure 6-13. This plan uses index access, and the execution time for this query is about two seconds.

Data Output	Explain	Messages	Notifications
	QUERY PLAN  text		
1	GroupAggregate (cost=1201.39..1205.93 rows=89 width=30)		
2	Group Key: bl.flight_id, f.departure_airport		
3	-> Sort (cost=1201.39..1202.03 rows=256 width=18)		
4	Sort Key: f.departure_airport		
5	-> Nested Loop (cost=6.54..1191.15 rows=256 width=18)		
6	-> Nested Loop (cost=6.11..1116.80 rows=89 width=26)		
7	-> Index Scan using flight_pkey on flight f (cost=0.42..8.44 rows=1 width=8)		
8	Index Cond: (flight_id = 222183)		
9	-> Nested Loop (cost=5.68..1107.47 rows=89 width=22)		
10	-> Bitmap Heap Scan on booking_leg bl (cost=5.25..355.42 rows=89 width=8)		
11	Recheck Cond: (flight_id = 222183)		
12	-> Bitmap Index Scan on booking_leg_flight_id (cost=0.00..5.23 rows=89 width=0)		
13	Index Cond: (flight_id = 222183)		
14	-> Index Scan using booking_pkey on booking b (cost=0.43..8.45 rows=1 width=14)		
15	Index Cond: (booking_id = bl.booking_id)		
16	-> Index Scan using passenger_booking_id on passenger p (cost=0.43..0.75 rows=9 width=8)		
17	Index Cond: (booking_id = b.booking_id)		

Figure 6-13. Execution plan for one flight

For all columns used in the GROUP BY clause, filtering should be pushed inside the grouping.

For PostgreSQL 12 and later, the optimizer takes care of this rewrite, but it may still be required in older versions.

Let's look at another example. Listing 6-17 calculates the same numbers (average price and number of customers) for all flights departing from ORD.

Listing 6-17. Select for multiple flights

```

SELECT
    flight_id,
    avg_price,
    num_passengers
FROM (SELECT
        b1.flight_id,
        departure_airport,
        (avg(price))::numeric (7,2) AS avg_price,
        count(DISTINCT passenger_id) AS num_passengers
FROM booking b
JOIN booking_leg b1 USING (booking_id)
JOIN flight f USING (flight_id)
JOIN passenger p USING (booking_id)
GROUP BY 1,2 )a WHERE departure_airport='ORD'

```

The execution plan for this query is shown in Figure 6-14. This query takes about 1.5 minutes to execute. It is a large query, and most of the joins are executed using the hash join algorithm. The important part is that the condition on `departure_airport` is applied first before the grouping.

	QUERY PLAN text
1	Subquery Scan on a (cost=899526.09..935621.26 rows=962538 width=26)
2	-> GroupAggregate (cost=899526.09..925995.88 rows=962538 width=30)
3	Group Key: bl.flight_id, f.departure_airport
4	-> Sort (cost=899526.09..901932.43 rows=962538 width=18)
5	Sort Key: bl.flight_id
6	-> Nested Loop (cost=9545.27..784126.73 rows=962538 width=18)
7	Join Filter: (b.booking_id = p.booking_id)
8	-> Nested Loop (cost=9544.84..532480.61 rows=334023 width=26)
9	-> Hash Join (cost=9544.41..367021.97 rows=334023 width=12)
10	Hash Cond: (bl.flight_id = f.flight_id)
11	-> Seq Scan on booking_leg bl (cost=0.00..310506.66 rows=17893566 width=8)
12	-> Hash (cost=9384.99..9384.99 rows=12753 width=8)
13	-> Bitmap Heap Scan on flight f (cost=243.26..9384.99 rows=12753 width=8)
14	Recheck Cond: (departure_airport = 'ORD'::bpchar)
15	-> Bitmap Index Scan on flight_departure_airport (cost=0.00..240.07 rows=12753 width=0)
16	Index Cond: (departure_airport = 'ORD'::bpchar)
17	-> Index Scan using booking_pkey on booking b (cost=0.43..0.50 rows=1 width=14)
18	Index Cond: (booking_id = bl.booking_id)
19	-> Index Scan using passenger_booking_id on passenger p (cost=0.43..0.64 rows=9 width=8)
20	Index Cond: (booking_id = bl.booking_id)

Figure 6-14. Execution plan for Listing 6-17

However, more complex filtering conditions can't be pushed inside grouping. Listing 6-18 calculates the same statistics, but the list of `flight_id` is not passed directly but is selected from the `booking_leg` table.

Listing 6-18. Condition can't be pushed inside the grouping

```
SELECT
    flight_id,
    avg_price,
    num_passengers
FROM (SELECT
        bl.flight_id,
        departure_airport,
```

```

        (avg(price))::numeric (7,2) AS avg_price,
        count(DISTINCT passenger_id) AS num_passengers
FROM booking b
JOIN booking_leg bl USING (booking_id)
JOIN flight f USING (flight_id)
JOIN passenger p USING (booking_id)
GROUP BY 1,2 )a
WHERE flight_id in
(SELECT
    flight_id
FROM flight
WHERE scheduled_departure BETWEEN '07-03-2023' AND '07-05-2023');
```

The execution plan (Figure 6-15) shows that grouping is done first and filtering is applied to the result of grouping. This means that, first, the calculations are performed for all flights in the system and then the subset is selected. The total execution time for this query is ten minutes.

	QUERY PLAN text
1	Merge Join (cost=12639813.37..14715429.51 rows=586141 width=26)
2	Merge Cond: (bl.flight_id = flight.flight_id)
3	-> GroupAggregate (cost=12630373.17..14053302.95 rows=51742901 width=30)
4	Group Key: bl.flight_id, f.departure_airport
5	-> Sort (cost=12630373.17..12759730.42 rows=51742901 width=18)
6	Sort Key: bl.flight_id, f.departure_airport
7	-> Hash Join (cost=1080689.84..2817439.82 rows=51742901 width=18)
8	Hash Cond: (b.booking_id = bl.booking_id)
9	-> Hash Join (cost=243101.21..775846.44 rows=16318749 width=22)
10	Hash Cond: (p.booking_id = b.booking_id)
11	-> Seq Scan on passenger p (cost=0.00..334860.49 rows=16318749 width=8)
12	-> Hash (cost=145003.98..145003.98 rows=5643298 width=14)
13	-> Seq Scan on booking b (cost=0.00..145003.98 rows=5643298 width=14)
14	-> Hash (cost=526548.06..526548.06 rows=17893566 width=12)
15	-> Hash Join (cost=26607.50..526548.06 rows=17893566 width=12)
16	Hash Cond: (bl.flight_id = f.flight_id)
17	-> Seq Scan on booking_leg bl (cost=0.00..310506.66 rows=17893566 width=8)
18	-> Hash (cost=15398.78..15398.78 rows=683178 width=8)
19	-> Seq Scan on flight f (cost=0.00..15398.78 rows=683178 width=8)
20	-> Sort (cost=9440.20..9459.54 rows=7739 width=4)
21	Sort Key: flight.flight_id
22	-> Bitmap Heap Scan on flight (cost=115.75..8940.34 rows=7739 width=4)
23	Recheck Cond: ((scheduled_departure >= '2023-07-03 00:00:00-05':timestamp with time zone) AND (scheduled_departure <= '2023-07-05 00:00:00-05':timestamp w
24	-> Bitmap Index Scan on flight_scheduled_departure (cost=0.00..113.81 rows=7739 width=0)
25	Index Cond: ((scheduled_departure >= '2023-07-03 00:00:00-05':timestamp with time zone) AND (scheduled_departure <= '2023-07-05 00:00:00-05':timestamp w

Figure 6-15. Execution plan for Listing 6-18

The query in Listing 6-18 is an example of what we call *pessimization*—using practices that guarantee slowing down the execution of a query. It’s easy to see why this query is written the way it is. First, a database developer figures out *how* to perform certain calculations or how to select specific values, and then they apply a filter to the result. Thus, they limit the optimizer to a certain order of operations, which in this case is not optimal.

Instead, the filtering can be done in the inner WHERE clause. When this change is made, there’s no longer a need for an inline SELECT—see Listing 6-19.

Listing 6-19. Condition is pushed inside grouping

```
SELECT bl.flight_id,
       departure_airport,
       (avg(price))::numeric (7,2) AS avg_price,
       count(DISTINCT passenger_id) AS num_passengers
FROM booking b
```



```
JOIN booking_leg bl USING (booking_id)
JOIN flight f USING (flight_id)
JOIN passenger p USING (booking_id)
    WHERE scheduled_departure
        BETWEEN '07-03-2023' AND '07-05-2023'
GROUP BY 1,2
```

The execution time is about one minute, and the execution plan is shown in Figure 6-16. This can be expressed as the generalization of the technique explained in the previous example.

	QUERY PLAN text
1	GroupAggregate (cost=682651.29..698770.14 rows=586140 width=30)
2	Group Key: bl.flight_id, f.departure_airport
3	-> Sort (cost=682651.29..684116.64 rows=586140 width=18)
4	Sort Key: bl.flight_id, f.departure_airport
5	-> Nested Loop (cost=9037.94..614473.98 rows=586140 width=18)
6	Join Filter: (b.booking_id = p.booking_id)
7	-> Nested Loop (cost=9037.51..462442.83 rows=202697 width=26)
8	-> Hash Join (cost=9037.07..366514.63 rows=202697 width=12)
9	Hash Cond: (bl.flight_id = f.flight_id)
10	-> Seq Scan on booking_leg bl (cost=0.00..310506.66 rows=17893566 width=8)
11	-> Hash (cost=8940.34..8940.34 rows=7739 width=8)
12	-> Bitmap Heap Scan on flight f (cost=115.75..8940.34 rows=7739 width=8)
13	Recheck Cond: ((scheduled_departure >= '2023-07-03 00:00:00-05':timestamp with time zone) AND (scheduled_departure <= '2023-07-05 00:00:00-05':timestamp w
14	-> Bitmap Index Scan on flight_scheduled_departure (cost=0.00..113.81 rows=7739 width=0)
15	Index Cond: ((scheduled_departure >= '2023-07-03 00:00:00-05':timestamp with time zone) AND (scheduled_departure <= '2023-07-05 00:00:00-05':timestamp w
16	-> Index Scan using booking_pkey on booking b (cost=0.43..0.47 rows=1 width=14)
17	Index Cond: (booking_id = bl.booking_id)
18	-> Index Scan using passenger_booking_id on passenger p (cost=0.43..0.64 rows=9 width=8)
19	Index Cond: (booking_id = bl.booking_id)

Figure 6-16. Execution plan with filtering pushed inside grouping

Filter rows are not needed for an aggregate prior to grouping.

Even the optimal execution of this query is not instantaneous, but it is the best we can achieve. Now is a good time to recall that optimization goals should be realistic. A long query on large data volumes can't be executed in a fraction of a second, even when executed optimally. The key is to use as few rows as necessary, but no fewer.

Grouping: Group First, Select Last

In some cases, the course of actions should be the opposite: GROUP BY should be executed as early as possible, followed by other operations. As you might have already guessed, this order of actions is desirable when grouping will reduce the size of the intermediate dataset.

The query in Listing 6-20 calculates the number of passengers departing from each city by month. In this case, it is not possible to reduce the number of rows needed, as all flights are used in the calculation.

Listing 6-20. Calculating number of passengers per city per month

```
SELECT
    city,
    date_trunc('month', scheduled_departure) AS month,
    count(*) passengers
FROM airport a
JOIN flight f ON airport_code = departure_airport
JOIN booking_leg l ON f.flight_id = l.flight_id
JOIN boarding_pass b ON b.booking_leg_id = l.booking_leg_id
GROUP BY 1,2
ORDER BY 3 DESC
```

The execution time for this query is 25 seconds, and the execution plan is in Figure 6-17.

	QUERY PLAN
	text
1	Sort (cost=11691560.21..11754793.94 rows=25293490 width=24)
2	Sort Key: (count(*)) DESC
3	-> GroupAggregate (cost=6456183.43..7025286.95 rows=25293490 width=24)
4	Group Key: a.city, (date_trunc('month',text,f.scheduled_departure))
5	-> Sort (cost=6456183.43..6519417.15 rows=25293490 width=16)
6	Sort Key: a.city, (date_trunc('month',text,f.scheduled_departure))
7	-> Hash Join (cost=804544.88..2049270.66 rows=25293490 width=16)
8	Hash Cond: (f.departure_airport = a.airport_code)
9	-> Hash Join (cost=804519.89..1919194.04 rows=25293490 width=12)
10	Hash Cond: (l.flight_id = f.flight_id)
11	-> Hash Join (cost=769001.43..1616337.75 rows=25293490 width=4)
12	Hash Cond: (b.booking_leg_id = l.booking_leg_id)
13	-> Seq Scan on boarding_pass b (cost=0.00..513692.90 rows=25293490 width=8)
14	-> Hash (cost=476508.08..476508.08 rows=17828108 width=8)
15	-> Seq Scan on booking_leg l (cost=0.00..476508.08 rows=17828108 width=8)
16	-> Hash (cost=23642.76..23642.76 rows=683176 width=16)
17	-> Seq Scan on flight f (cost=0.00..23642.76 rows=683176 width=16)
18	-> Hash (cost=16.66..16.66 rows=666 width=12)
19	-> Seq Scan on airport a (cost=0.00..16.66 rows=666 width=12)

Figure 6-17. Execution plan with grouping done last

The execution of this query is significantly improved with a nontrivial rewrite, as shown in Listing 6-21.

Listing 6-21. Query rewrite that forces grouping be done first

```
SELECT
    city,
    date_trunc('month', scheduled_departure),
    sum(passengers) passengers
FROM airport a
JOIN flight f ON airport_code = departure_airport
JOIN (
    SELECT flight_id, count(*) passengers
    FROM booking_leg l
    JOIN boarding_pass b USING (booking_leg_id)
    GROUP BY flight_id
) cnt
```

```

USING (flight_id)
GROUP BY 1,2
ORDER BY 3 DESC

```

What is happening here? First, the number of departing passengers is summed for each flight in the inline view `cnt`. After, the result is joined with the `flight` table to retrieve airport code and then joined with the `airport` table to find the city where each airport is located. After this, the flight totals are summed by city. This way, the execution time is 15 seconds. The execution plan is shown in Figure 6-18.

QUERY PLAN		text	
1	2		
1	Sort	(cost=3165092.63..3165617.34 rows=209882 width=49)	
2	Sort Key:	(sum(cnt.passengers)) DESC	
3	-> GroupAggregate	(cost=3134117.83..3139364.88 rows=209882 width=49)	
4	Group Key:	a.city, (date_trunc('month':text, f.scheduled_departure))	
5	-> Sort	(cost=3134117.83..3134642.54 rows=209882 width=25)	
6	Sort Key:	a.city, (date_trunc('month':text, f.scheduled_departure))	
7	-> Hash Join	(cost=2901723.58..3110542.59 rows=209882 width=25)	
8	Hash Cond:	(f.departure_airport = a.airport_code)	
9	-> Hash Join	(cost=2901698.01..3109438.00 rows=209882 width=20)	
10	Hash Cond:	(cnt.flight_id = f.flight_id)	
11	-> Subquery Scan on cnt	(cost=2874423.51..3076226.55 rows=209882 width=12)	
12	-> HashAggregate	(cost=2874423.51..3074127.73 rows=209882 width=12)	
13	Group Key:	l.flight_id	
14	Planned Partitions:	4	
15	-> Hash Join	(cost=604073.24..1451664.58 rows=25293492 width=4)	
16	Hash Cond:	(b.booking_leg_id = l.booking_leg_id)	
17	-> Seq Scan on boarding_pass b	(cost=0.00..513692.92 rows=25293492 width=...	
18	-> Hash	(cost=310506.66..310506.66 rows=17893566 width=8)	
19	-> Seq Scan on booking_leg l	(cost=0.00..310506.66 rows=17893566 width=...	
20	-> Hash	(cost=15398.78..15398.78 rows=683178 width=16)	
21	-> Seq Scan on flight f	(cost=0.00..15398.78 rows=683178 width=16)	
22	-> Hash	(cost=16.92..16.92 rows=692 width=13)	
23	-> Seq Scan on airport a	(cost=0.00..16.92 rows=692 width=13)	

Figure 6-18. Execution plan with grouping forced to be first

Using SET Operations

We rarely use set theoretical operations (UNION, EXCEPT, etc.) in SQL queries. For large queries, however, these operations may prompt the optimizer to choose more efficient algorithms.

Use set operations to (sometimes) prompt an alternative execution plan and improve readability.

Often, we can

- Use EXCEPT instead of NOT EXISTS and NOT IN.
- Use INTERSECT instead of EXISTS and IN.
- Use UNION instead of complex selection criteria with OR.

Sometimes, there can be significant performance gains, and sometimes the execution time changes only slightly, but the code becomes cleaner and easier to maintain. Listing 6-22 shows a rewrite of the query in Listing 6-8, returning flights with no bookings.

Listing 6-22. Using EXCEPT instead of NOT IN

```
SELECT
    flight_id
FROM flight f
EXCEPT
SELECT
    flight_id
FROM booking_leg
```

Execution time is one minute and three seconds, which is almost twice as fast as an anti-join.

The execution plan with the EXCEPT operation is shown in Figure 6-19.

QUERY PLAN	
Read-only column	
1	SetOp Except (cost=3518478.09..3611034.51 rows=683176 width=8)
2	-> Sort (cost=3518478.09..3564756.30 rows=18511284 width=8)
3	Sort Key: <i>"*SELECT* 1".flight_id</i>
4	-> Append (cost=0.00..777820.10 rows=18511284 width=8)
5	-> Subquery Scan on <i>"*SELECT* 1"</i> (cost=0.00..30474.52 rows=683176 width=8)
6	-> Seq Scan on flight f (cost=0.00..23642.76 rows=683176 width=4)
7	-> Subquery Scan on <i>"*SELECT* 2"</i> (cost=0.00..654789.16 rows=17828108 width=8)
8	-> Seq Scan on booking_leg (cost=0.00..476508.08 rows=17828108 width=4)

Figure 6-19. Execution plan with EXCEPT

Listing 6-23 shows a rewrite of the query in Listing 6-4 using set operations, showing all flights with a booking.

Listing 6-23. Using INTERSECT instead of IN

```
SELECT
    flight_id
FROM flight f
INTERSECT
SELECT
    flight_id
FROM booking_leg
```

The execution time of this query is 49 seconds. This is less than the version of the query using the IN keyword and approximately equal to the runtime of the query with an index-only scan (see Listing 6-10). The execution plan is shown in Figure 6-20.

	QUERY PLAN text
1	HashSetOp Intersect (cost=0.00..824098.31 rows=139576 width=8)
2	-> Append (cost=0.00..777820.10 rows=18511284 width=8)
3	-> Subquery Scan on "*SELECT* 2" (cost=0.00..654789.16 rows=17828108 width=8)
4	-> Seq Scan on booking_leg (cost=0.00..476508.08 rows=17828108 width=4)
5	-> Subquery Scan on "*SELECT* 1" (cost=0.00..30474.52 rows=683176 width=8)
6	-> Seq Scan on flight f (cost=0.00..23642.76 rows=683176 width=4)

Figure 6-20. Execution plan with INTERSECT

We rarely need to rewrite complex selection criteria with OR into set theoretical UNION ALL, because most of the time the PostgreSQL optimizer does a decent job analyzing such criteria and making use of all suitable indexes. However, sometimes rewriting this way makes code more maintainable, especially when the query contains a large number of different selection criteria connected with OR. Listing 6-24 is a query that calculates the number of passengers on delayed flights from FRA using two different sets of selection criteria. The first group is passengers on flights delayed by more than an hour, with changes to the boarding pass more than 30 minutes after the scheduled departure. The second is passengers on flights delayed by more than a half hour but less than an hour.

Listing 6-24. Query with complex selection criteria with OR

```
SELECT
    CASE
        WHEN actual_departure>scheduled_departure + interval '1 hour'
            THEN 'Late group 1'
        ELSE 'Late group 2'
    END AS grouping,
    flight_id,
    count(*) AS num_passengers
FROM boarding_pass bp
JOIN booking_leg bl USING (booking_leg_id)
JOIN booking b USING (booking_id)
JOIN flight f USING (flight_id)
WHERE departure_airport='FRA'
    AND actual_departure>'2023-07-01'
```

```

AND
(
    (actual_departure>scheduled_departure + interval '30 minute'
      AND actual_departure<=scheduled_departure + interval '1 hour'
    )
    OR
    (actual_departure>scheduled_departure + interval '1 hour'
      AND bp.update_ts >scheduled_departure + interval '30 minute'
    )
)
GROUP BY 1,2

```

The rewrite of this query using UNION ALL is shown in Listing 6-25. The execution time difference is not significant (about three seconds), but the code is more maintainable.

Listing 6-25. Rewrite of a complex condition with OR using UNION ALL

```

SELECT
    'Late group 1' AS grouping,
    flight_id,
    count(*) AS num_passengers
FROM boarding_pass bp
JOIN booking_leg bl USING (booking_leg_id)
JOIN booking b USING (booking_id)
JOIN flight f USING (flight_id)
WHERE departure_airport='FRA'
    AND actual_departure>scheduled_departure + interval '1 hour'
    AND bp.update_ts > scheduled_departure + interval '30 minutes'
    AND actual_departure>'2023-07-01'
GROUP BY 1,2
UNION ALL
SELECT
    'Late group 2' AS grouping,
    flight_id,
    count(*) AS num_passengers
FROM boarding_pass bp

```



```

JOIN booking_leg bl USING(booking_leg_id)
JOIN booking b USING (booking_id)
JOIN flight f USING (flight_id)
WHERE departure_airport='FRA'
      AND actual_departure>scheduled_departure + interval '30 minute'
      AND actual_departure<=scheduled_departure + interval '1 hour'
      AND actual_departure>'2023-07-01'
GROUP BY 1,2

```

It's worth noting that with large queries you always need to take into consideration how much RAM you have available. For both hash joins and set theoretical operations, if the participating datasets can't fit into main memory, the execution speed increases significantly.

Avoiding Multiple Scans

Another source of slowness in long queries is the presence of multiple table scans. This common problem is the direct result of imperfect design. Designs can be fixed, at least theoretically. But since we often find ourselves in situations where we can't control the design, we are going to suggest ways to write performant queries even on an imperfect schema.

The situation that we are modeling in our `postgres_air` schema is not uncommon in the real world. The system is already up and running, and all of a sudden, we need to store some additional information for objects that are already present in the database.

For the past 30 years, the easiest solution in such cases is to use an entity-attribute-value (EAV) table, which can store arbitrary attributes—those needed now and any that will eventually be needed. In the `postgres_air` schema, this pattern is implemented in the table `custom_field`. For each passenger, a passport number, a passport expiration date, and the country that issued the passport are stored. The attributes are accordingly named `'passport_num'`, `'passport_exp_date'`, and `'passport_country'`.

This table is not included into the `postgres_air` distribution. To run the example locally, execute the following script from the `postgres_air` GitHub repository:

https://github.com/hettie-d/postgres_air/blob/main/tables/custom_field.sql

Now, imagine a request for a report that lists passenger names and their passport information. Listing 6-26 is a typical suggested solution: the table `custom_field` is scanned three times! In order to avoid spill to the disk, passengers are limited to the first five million, which allows us to show the true ratio of execution times. The execution plan in Figure 6-21 confirms three table scans, and the execution time for this query is five minutes.

Listing 6-26. Multiple scans of a large table

```
SELECT
    first_name,
    last_name,
    pn.custom_field_value AS passport_num,
    pe.custom_field_value AS passport_exp_date,
    pc.custom_field_value AS passport_country
FROM passenger p
JOIN custom_field pn ON pn.passenger_id=p.passenger_id
                        AND pn.custom_field_name='passport_num'
JOIN custom_field pe ON pe.passenger_id=p.passenger_id
                        AND pe.custom_field_name='passport_
                        exp_date'
JOIN custom_field pc ON pc.passenger_id=p.passenger_id
                        AND pc.custom_field_name='passport_
                        country'
WHERE p.passenger_id<5000000
```

QUERY PLAN	
	text
1	Hash Join (cost=3344203.17..4769951.67 rows=4306126 width=51)
2	Hash Cond: (pe.passenger_id = pn.passenger_id)
3	-> Seq Scan on custom_field pe (cost=0.00..1022564.75 rows=16454200 width=17)
4	Filter: (custom_field_name = 'passport_exp_date':text)
5	-> Hash (cost=3244130.53..3244130.53 rows=4494451 width=50)
6	-> Hash Join (cost=1806471.88..3244130.53 rows=4494451 width=50)
7	Hash Cond: (pc.passenger_id = pn.passenger_id)
8	-> Seq Scan on custom_field pc (cost=0.00..1022564.75 rows=16273188 width=17)
9	Filter: (custom_field_name = 'passport_country':text)
10	-> Hash (cost=1710124.97..1710124.97 rows=4743193 width=33)
11	-> Hash Join (cost=430701.56..1710124.97 rows=4743193 width=33)
12	Hash Cond: (pn.passenger_id = p.passenger_id)
13	-> Seq Scan on custom_field pn (cost=0.00..1022564.75 rows=16194911 width=17)
14	Filter: (custom_field_name = 'passport_num':text)
15	-> Hash (cost=343266.86..343266.86 rows=5029896 width=16)
16	-> Seq Scan on passenger p (cost=0.00..343266.86 rows=5029896 width=16)
17	Filter: (passenger_id < 5000000)

Figure 6-21. Execution plan with multiple scans

Scanning this table three times is like sorting apples, oranges, and lemons from one black box into three buckets, doing so by first sorting out all the apples, returning all the oranges and lemons back into the box, then sorting the oranges, and then finally returning to the box for the lemons. A more effective way to do this job would be to place all three buckets before you and sort each fruit into the correct bucket when first removing it from the black box.

When retrieving multiple attributes from an entity-attribute-value table, join to the table only once and use FILTER in the aggregate function MAX() in SELECT list to return the appropriate values in each column.

To replicate this effect on the custom_field table, the query can be rewritten as shown in Listing 6-27.

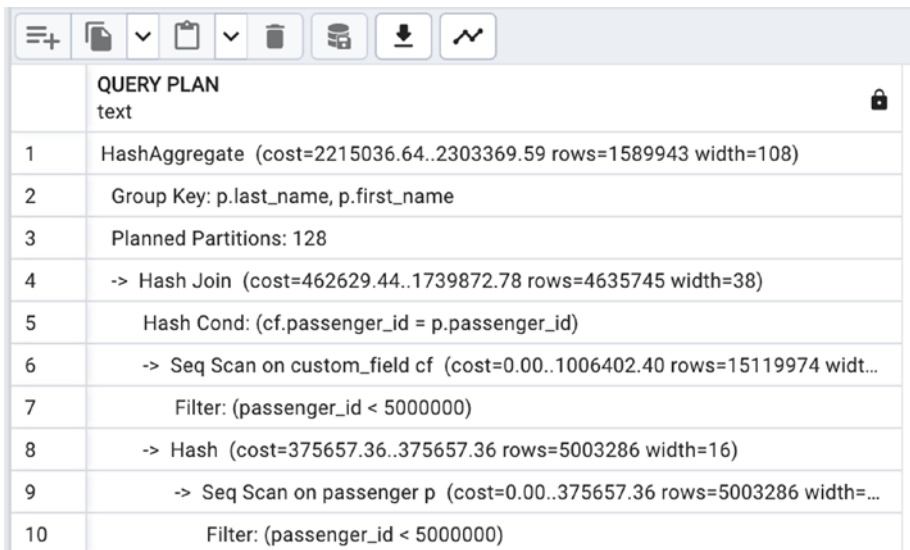
Listing 6-27. One table scan to retrieve multiple attributes

```

SELECT
  last_name,
  first_name,
  coalesce(max (custom_field_value )
            FILTER (WHERE custom_field_name ='passport_num' ),'')
            AS passport_num,
  coalesce(max (custom_field_value )
            FILTER (WHERE custom_field_name ='passport_exp_date' ),'')
            AS passport_exp_date,
  coalesce(max (custom_field_value )
            FILTER (WHERE custom_field_name ='passport_country' ),'')
            AS passport_country
FROM passenger p
JOIN custom_field cf  USING (passenger_id)
WHERE cf.passenger_id<5000000
      AND p.passenger_id<5000000
GROUP by 1,2;

```

The execution plan for Listing 6-27 is shown in Figure 6-22.



QUERY PLAN	
	text
1	HashAggregate (cost=2215036.64..2303369.59 rows=1589943 width=108)
2	Group Key: p.last_name, p.first_name
3	Planned Partitions: 128
4	-> Hash Join (cost=462629.44..1739872.78 rows=4635745 width=38)
5	Hash Cond: (cf.passenger_id = p.passenger_id)
6	-> Seq Scan on custom_field cf (cost=0.00..1006402.40 rows=15119974 width=...
7	Filter: (passenger_id < 5000000)
8	-> Hash (cost=375657.36..375657.36 rows=5003286 width=16)
9	-> Seq Scan on passenger p (cost=0.00..375657.36 rows=5003286 width=...
10	Filter: (passenger_id < 5000000)

Figure 6-22. One table scan

This looks much better—only one table scan—except that when you try to execute it, it will run significantly longer. A closer look shows why: there may be many passengers with the same first and last names, so not only does it take longer but the result is also incorrect. Let’s modify the query one more time—see Listing 6-28.

Listing 6-28. Correction for the query in Listing 6-27

```
SELECT
  last_name,
  first_name,
  p.passenger_id,
  coalesce(max (custom_field_value )
            FILTER (WHERE custom_field_name ='passport_num' ),'')
            AS passport_num,
  coalesce(max (custom_field_value )
            FILTER (WHERE custom_field_name ='passport_exp_date' ),'')
            AS passport_exp_date,
  coalesce(max (custom_field_value )
            FILTER (WHERE custom_field_name ='passport_country' ),'')
            AS passport_country
FROM passenger p
JOIN custom_field cf USING (passenger_id)
WHERE cf.passenger_id<5000000
      AND p.passenger_id<5000000
GROUP by 3,1,2;
```

The execution plan in Figure 6-23 looks much better—the grouping column is now `passenger_id`.

	QUERY PLAN	
	text	
1	HashAggregate (cost=2239664.04..2367509.19 rows=4635745 width=112)	
2	Group Key: p.passenger_id	
3	Planned Partitions: 256	
4	-> Hash Join (cost=462629.44..1739872.78 rows=4635745 width=42)	
5	Hash Cond: (cf.passenger_id = p.passenger_id)	
6	-> Seq Scan on custom_field cf (cost=0.00..1006402.40 rows=15119974 width=...	
7	Filter: (passenger_id < 5000000)	
8	-> Hash (cost=375657.36..375657.36 rows=5003286 width=16)	
9	-> Seq Scan on passenger p (cost=0.00..375657.36 rows=5003286 width=...	
10	Filter: (passenger_id < 5000000)	

Figure 6-23. Execution plan for Listing 6-28

There's one more optimization to make here; attributes from an EAV table are often joined to other tables, and we can reduce the size of the intermediate result set by “collapsing” and filtering this table to the needed values before executing other joins. This is a more specific case of the generalized technique of grouping before joining earlier.

Pull values from an EAV table into a subquery before joining to other tables.

Doing this for the passport example, we can modify the query one more time, as shown in Listing 6-29.

Listing 6-29. Moving grouping to the subquery

```
SELECT
    last_name,
    first_name,
    passport_num,
```

```

        passport_exp_date,
        passport_country
FROM   passenger p
JOIN (
SELECT
    cf.passenger_id,
    coalesce(max (custom_field_value )
              FILTER (WHERE custom_field_name = 'passport_num' ), '')
              AS passport_num,
    coalesce(max (custom_field_value )
              FILTER (WHERE custom_field_name = 'passport_exp_date' ), '')
              AS passport_exp_date,
    coalesce(max (custom_field_value )
              FILTER (WHERE custom_field_name = 'passport_country' ), '')
              AS passport_country
FROM   custom_field cf
WHERE  cf.passenger_id<5000000
GROUP BY 1
) info USING (passenger_id)
WHERE  p.passenger_id<5000000

```

The execution plan is shown in Figure 6-24.

QUERY PLAN	
	text
1	Hash Join (cost=3995605.13..4895910.34 rows=3263588 width=108)
2	Hash Cond: (info.passenger_id = p.passenger_id)
3	-> Subquery Scan on info (cost=3532975.70..4048265.98 rows=10644540 width=100)
4	-> GroupAggregate (cost=3532975.70..3941820.58 rows=10644540 width=100)
5	Group Key: cf.passenger_id
6	-> Sort (cost=3532975.70..3570775.63 rows=15119974 width=30)
7	Sort Key: cf.passenger_id
8	-> Seq Scan on custom_field cf (cost=0.00..1006402.40 rows=15119974 width=...
9	Filter: (passenger_id < 5000000)
10	-> Hash (cost=375657.36..375657.36 rows=5003286 width=16)
11	-> Seq Scan on passenger p (cost=0.00..375657.36 rows=5003286 width=16)
12	Filter: (passenger_id < 5000000)

Figure 6-24. Execution plan with grouping moved to the subquery

Conclusion

This chapter formally defined long queries and explored optimization techniques for them.

The first important principle of this chapter is that indexes do not necessarily make queries run faster and can, in fact, make a long query run slower. A common misconception is that if no indexes can be built, there is nothing you can do to optimize a full table scan. Hopefully, this chapter has definitively demonstrated that there are multiple possibilities to optimize a full table scan.

As with short queries, long queries are optimized by reducing the size of intermediate results and doing the necessary work on as few rows as possible. In the case of short queries, this is accomplished by applying indexes on the most restrictive criteria. In the case of long queries, this is accomplished by being mindful of join order, applying semi- and anti-joins, and filtering before grouping, grouping before joining, and applying set operations.

CHAPTER 7

Long Queries: Additional Techniques

Chapter 6 discussed multiple ways of improving the performance of long queries. So far, every covered technique related to rewriting queries without creating any additional database objects. This chapter addresses additional ways of improving long query performance, including different ways to materialize intermediate results. Temporary tables, CTEs (common table expressions), views, and materialized views are discussed—when each tool can improve performance and how each can be abused and lead to performance degradation. Finally, this chapter covers partitioning and parallel execution.

Structuring Queries

Readers familiar with object-oriented programming (OOP) will be familiar with the concepts of decomposition (factoring) and encapsulation. OOP best practices dictate that code should be decomposed (or factored) into many smaller classes and objects responsible for a well-defined subset of system behavior, as well as encapsulated, restricting direct access to components and thus obscuring their implementation. These two principles make application code more readable and more manageable and make it easier to make changes.

Coming from this paradigm, when one is confronted with a single `SELECT` statement of 500+ lines, it's understandably tempting to apply these same principles, factoring the code into smaller pieces and encapsulating some of the logic.

However, SQL's declarative nature dictates a very different style of factoring SQL queries from what would be used with application code. In SQL, as in any language, code should be kept easy to understand and modify, but not at the expense of performance.

We can approach factoring and encapsulation in a variety of ways in SQL, each with its own advantages and pitfalls. Some are used (with varying effectiveness) to improve performance and store intermediate results. Others are used to make code reusable. Others impact the way data is stored. This chapter covers several approaches, and others, such as functions, will be covered in depth in subsequent chapters.

In any case, any decomposition or encapsulation should correspond to a logical entity—for example, a report or a daily refresh.

Temporary Tables and CTEs

In Chapter 6, we mentioned that sometimes the attempts of SQL developers to speed up query execution may result in slowing execution. This often happens when they decide to use *temporary tables*.

Temporary Tables

To create a temporary table, one executes a regular `create table` statement, adding the keyword `temporary` or just `temp`:

```
CREATE TEMP TABLE interim_results
```

Temporary tables are visible to the current session only and are dropped when the session disconnects if not dropped explicitly before that. Otherwise, they are as good as regular tables, can be used in the queries with no limitations, and can even be indexed. Temporary tables are often used to store intermediate results of queries, so the `CREATE` statement often looks like

```
CREATE TEMP TABLE interim_results AS
SELECT ...
```

All of this looks very convenient, so what's wrong with this approach?

It all works great if you use a temporary table to store results of your query for some analysis and then discard it when done. But often, when a SQL developer starts to use temporary tables to store the results of each step, the code starts to look like this:

```
CREATE TEMP TABLE T1 AS SELECT <...> ;
CREATE TEMP TABLE T2 AS SELECT <...>
```

```
FROM T1
      INNER JOIN <...>
<...>
```

The chain of temporary tables can become quite long. Does it cause any problems? Yes, and there are many of them, including the following:

- *Indexes* – After selected data is stored in a temporary table, we can’t use indexes that were created on the source table(s). We either need to continue without indexes or build new ones on temporary tables, which takes time.
- *Statistics* – Since we created a new table, the optimizer can’t use statistical data on value distribution from the source table(s), so we need either to go without statistics or run the ANALYZE command on the temporary table.
- *Disk space* – Temporary tables are still tables that are stored on disk, so when they are overused, significant database size growth is possible.
- *Excessive I/O* – Also, since they are tables written to disk, it takes extra time to write to and read from disk.

The most important negative implication of excessive use of temporary tables is that this practice blocks the optimizer from doing rewrites.

By saving the results of each join in a temporary table, you prevent the optimizer from choosing the optimal join order; you “lock” in the order in which you created the temporary tables.

When we looked at the execution plan of the query in Listing 6-15, we observed that PostgreSQL was able to push the filtering condition inside grouping. What would happen if a temp table was created for intermediate results?

Listing 7-1. Inefficient usage of temp tables

```
CREATE TEMP TABLE flights_totals AS
SELECT
    bl.flight_id,
    departure_airport,
    (avg(price))::numeric (7,2) AS avg_price,
    count(DISTINCT passenger_id) AS num_passengers
```

```

FROM booking b
JOIN booking_leg bl USING (booking_id)
JOIN flight f USING (flight_id)
JOIN passenger p USING (booking_id)
GROUP BY 1,2;

SELECT
    flight_id,
    avg_price,
    num_passengers
FROM flights_totals
WHERE departure_airport='ORD';

```

Creating the temporary table took 50 seconds and produced over 500,000 rows, out of which we needed just 10,000. At the same time, the query in Listing 6-15 took just 0.1 seconds to execute.

Common Table Expressions (CTEs)

If temporary tables are dangerous, should a *CTE* (*common table expression*) be used instead? First, what are CTEs?

Common table expressions, or CTEs, can be thought of as defining temporary tables that exist just for one query. Each auxiliary statement in a WITH clause can be a SELECT, INSERT, UPDATE, or DELETE; and the WITH clause itself is attached to a primary statement that can also be a SELECT, INSERT, UPDATE, or DELETE.

Let's give a CTE a try. In Listing 7-2, the query from Listing 7-1 is modified to use a CTE instead of a temporary table.

Listing 7-2. Example of a query with a CTE

```

WITH flights_totals AS(
    SELECT
        bl.flight_id,
        departure_airport,
        (avg(price))::numeric (7,2) AS avg_price,

```

```

        count(DISTINCT passenger_id) AS num_passengers
FROM booking b
JOIN booking_leg bl USING (booking_id)
JOIN flight f USING (flight_id)
JOIN passenger p USING (booking_id)
GROUP BY 1,2)
SELECT
    flight_id,
    avg_price,
    num_passengers
FROM flights_totals
WHERE departure_airport='ORD'
```

What you will see in the execution plan depends on whether you are running a PostgreSQL version below 12 or 12 and above. For all versions below 12, a CTE was processed exactly like a temporary table. The results were materialized in main memory with possible disk failover. That means that there was no advantage to using a CTE instead of a temporary table.

To be fair, a CTE's intended purpose was different. The idea behind a CTE was that if you needed to use some (possibly complex) subselect more than once, you could define it as a CTE and reference it in a query multiple times. In this case, PostgreSQL will compute results just once and reuse them as many times as needed.

Because of this intended usage, the optimizer planned CTE execution separately from the rest of the query and did not push any join conditions inside the CTE, providing a so-called *optimization fence*. This is especially important if `WITH` is used in `INSERT/DELETE/UPDATE` statements where there may be side effects or in recursive CTE calls. In addition, having the optimization fence means that the tables involved in the CTE are not counted against `join_collapse_limit`. This allows us to effectively use PostgreSQL optimizer capabilities with queries that join a large number of tables.

For the query in Listing 7-2, in PostgreSQL versions before 12, the CTE `flight_totals` would be calculated for all flights, and only after that a subset of flights would be selected.

PostgreSQL 12 brought a drastic change to CTE optimization. For `SELECT` statements with no recursion, if a CTE is used in a query only once, it will be inlined into the outer query (removing the optimization fence). If it is called more than once, the old behavior will be preserved.

What is more important, the behavior described earlier is a default, but it can be overwritten by using the keywords `MATERIALIZED` and `NOT MATERIALIZED`. (See Listing 7-3.) The first one forces the old behavior, and the second one forces inlining, regardless of all other considerations.

Listing 7-3. Usage of the `MATERIALIZED` keyword

```
WITH flights_totals AS MATERIALIZED (
    SELECT
        bl.flight_id,
        departure_airport,
        (avg(price))::numeric (7,2) AS avg_price,
        count(DISTINCT passenger_id) AS num_passengers
    FROM booking b
    JOIN booking_leg bl USING (booking_id)
    JOIN flight f USING (flight_id)
    JOIN passenger p USING (booking_id)
    GROUP BY 1,2)
SELECT
    flight_id,
    avg_price,
    num_passengers
FROM flights_totals
WHERE departure_airport='ORD'
```

Figure 7-1 presents the execution plan for Listing 7-2, the way it works in PostgreSQL 12. If the keyword `MATERIALIZED` is added, as shown in Listing 7-3, the old behavior will be forced, as shown in Figure 7-2.

	QUERY PLAN text
1	Subquery Scan on flights_totals (cost=899526.09..935621.26 rows=962538 width=26)
2	-> GroupAggregate (cost=899526.09..925995.88 rows=962538 width=30)
3	Group Key: bl.flight_id, f.departure_airport
4	-> Sort (cost=899526.09..901932.43 rows=962538 width=18)
5	Sort Key: bl.flight_id
6	-> Nested Loop (cost=9545.27..784126.73 rows=962538 width=18)
7	Join Filter: (b.booking_id = p.booking_id)
8	-> Nested Loop (cost=9544.84..532480.61 rows=334023 width=26)
9	-> Hash Join (cost=9544.41..367021.97 rows=334023 width=12)
10	Hash Cond: (bl.flight_id = f.flight_id)
11	-> Seq Scan on booking_leg bl (cost=0.00..310506.66 rows=17893566 width=8)
12	-> Hash (cost=9384.99..9384.99 rows=12753 width=8)
13	-> Bitmap Heap Scan on flight f (cost=243.26..9384.99 rows=12753 width=8)
14	Recheck Cond: (departure_airport = 'ORD'::bpchar)
15	-> Bitmap Index Scan on flight_departure_airport (cost=0.00..240.07 rows=12753 width=0)
16	Index Cond: (departure_airport = 'ORD'::bpchar)
17	-> Index Scan using booking_pkey on booking b (cost=0.43..0.50 rows=1 width=14)
18	Index Cond: (booking_id = bl.booking_id)
19	-> Index Scan using passenger_booking_id on passenger p (cost=0.43..0.64 rows=9 width=8)
20	Index Cond: (booking_id = bl.booking_id)

Figure 7-1. Execution plan for a CTE with inlining

	QUERY PLAN
	text
1	CTE Scan on flights_totals (cost=14081839.16..15242007.67 rows=257815 width=26)
2	Filter: (departure_airport = 'ORD'::bpchar)
3	CTE flights_totals
4	-> GroupAggregate (cost=12663855.42..14081839.16 rows=51563045 width=30)
5	Group Key: bl.flight_id, f.departure_airport
6	-> Sort (cost=12663855.42..12792763.04 rows=51563045 width=18)
7	Sort Key: bl.flight_id, f.departure_airport
8	-> Hash Join (cost=1151589.89..2886328.12 rows=51563045 width=18)
9	Hash Cond: (b.booking_id = bl.booking_id)
10	-> Hash Join (cost=314001.30..846710.40 rows=16313907 width=22)
11	Hash Cond: (p.booking_id = b.booking_id)
12	-> Seq Scan on passenger p (cost=0.00..334787.07 rows=16313907 width=8)
13	-> Hash (cost=215591.02..215591.02 rows=5661302 width=14)
14	-> Seq Scan on booking b (cost=0.00..215591.02 rows=5661302 width=14)
15	-> Hash (cost=526548.02..526548.02 rows=17893566 width=12)
16	-> Hash Join (cost=26607.46..526548.02 rows=17893566 width=12)
17	Hash Cond: (bl.flight_id = f.flight_id)
18	-> Seq Scan on booking_leg bl (cost=0.00..310506.66 rows=17893566 width=8)
19	-> Hash (cost=15398.76..15398.76 rows=683176 width=8)
20	-> Seq Scan on flight f (cost=0.00..15398.76 rows=683176 width=8)

Figure 7-2. Forced materialization of a CTE

Prior to these recent changes, we would discourage SQL developers from using multiple embedded CTEs, when a SQL statement would look like this:

```
WITH z AS (  
  WITH y AS (  
    WITH x AS (SELECT ...)  
    SELECT ... FROM a JOIN x ...  
    SELECT ... FROM b  
    JOIN y...)
```



```
SELECT ... FROM c
      JOIN z
...

```

However, with the changes introduced in PostgreSQL 12, such queries are much more manageable. We would still encourage SQL developers to be mindful not to force a suboptimal execution plan, but using a chain of CTEs is much better than using a sequence of temporary tables; in the latter case, the optimizer is helpless.

To conclude, it's important to mention that there are situations where storing intermediate results is beneficial. However, almost always there are better ways than using temporary tables. We will discuss other options later in this chapter.

Views: To Use or Not to Use

Views are the most controversial database object. They seem to be easy to understand, and the advantages of creating a view seem so obvious. Why might they cause problems?

Although we are sure that most readers have had a chance to create at least a couple of views for some project, let's give a formal definition. The simplest definition is

A view is a database object that stores a query that defines a virtual table.

A view is a virtual table in the sense that syntactically, views may be used in a `SELECT` statement in the same way as a table. However, they differ significantly from tables in that no data is stored; only the query that defines the view is stored in the database.

Let's take another look at the query in Listing 6-14. This query calculates the totals for all flights in the `postgres_air` schema, but we want to use this query logic to select the totals for specific flights and/or departure airports. Listing 7-4 creates a view that encapsulates this logic.

Listing 7-4. Create a view

```
CREATE VIEW flight_stats AS
SELECT
    bl.flight_id,
```

```

        departure_airport,
        (avg(price))::numeric (7,2) AS avg_price,
        count(DISTINCT passenger_id) AS num_passengers
FROM booking b
JOIN booking_leg bl USING (booking_id)
JOIN flight f USING (flight_id)
JOIN passenger p USING (booking_id)
GROUP BY 1,2

```

Now it is easy to select flight statistics for any particular flight:

```

SELECT *
FROM flight_stats
WHERE flight_id=222183

```

This query plan looks identical to the plan in Figure 6-13. The reason is that in the first step of query processing, the query parser transforms views into inline subqueries. In this case, this works to our advantage, since the filtering condition is pushed inside the grouping. But if a nonconstant search criterion is used, the results might be disappointing. In Listing 7-5, flight statistics from the view `flight_stats` are limited by the flight's departure date.

Listing 7-5. Query using the view

```

SELECT *
FROM flight_stats fs
JOIN (SELECT
        flight_id
      FROM flight f
      WHERE actual_departure between '2023-08-01' and '2023-08-14'
    ) fl
ON fl.flight_id=fs.flight_id

```

The execution plan for this query is shown in Figure 7-3.

	QUERY PLAN text
1	Hash Join (cost=12641030.68..14717215.42 rows=3663700 width=34)
2	Hash Cond: (bl.flight_id = f.flight_id)
3	-> GroupAggregate (cost=12630373.17..14053302.95 rows=51742901 width=30)
4	Group Key: bl.flight_id, f_1.departure_airport
5	-> Sort (cost=12630373.17..12759730.42 rows=51742901 width=18)
6	Sort Key: bl.flight_id, f_1.departure_airport
7	-> Hash Join (cost=1080689.84..2817439.82 rows=51742901 width=18)
8	Hash Cond: (b.booking_id = bl.booking_id)
9	-> Hash Join (cost=243101.21..775846.44 rows=16318749 width=22)
10	Hash Cond: (p.booking_id = b.booking_id)
11	-> Seq Scan on passenger p (cost=0.00..334860.49 rows=16318749 width=8)
12	-> Hash (cost=145003.98..145003.98 rows=5643298 width=14)
13	-> Seq Scan on booking b (cost=0.00..145003.98 rows=5643298 width=14)
14	-> Hash (cost=526548.06..526548.06 rows=17893566 width=12)
15	-> Hash Join (cost=26607.50..526548.06 rows=17893566 width=12)
16	Hash Cond: (bl.flight_id = f_1.flight_id)
17	-> Seq Scan on booking_leg bl (cost=0.00..310506.66 rows=17893566 width=8)
18	-> Hash (cost=15398.78..15398.78 rows=683178 width=8)
19	-> Seq Scan on flight f_1 (cost=0.00..15398.78 rows=683178 width=8)
20	-> Hash (cost=10052.84..10052.84 rows=48373 width=4)
21	-> Bitmap Heap Scan on flight f (cost=760.25..10052.84 rows=48373 width=4)
22	Recheck Cond: ((actual_departure >= '2023-08-01 00:00:00-05':timestamp with time zone) AND (actual_departure <= '2023-08-14 00:00:00-05':timestar
23	-> Bitmap Index Scan on flight_actual_departure (cost=0.00..748.15 rows=48373 width=0)
24	Index Cond: ((actual_departure >= '2023-08-01 00:00:00-05':timestamp with time zone) AND (actual_departure <= '2023-08-14 00:00:00-05':timestar

Figure 7-3. Execution plan in which the condition can't be pushed

Looking at this execution plan, we observe that first, the statistics for all flights are calculated, and only after that are the results joined with selected flights. The execution time of this query is ten minutes.

Without using the view, we follow the pattern explained in Chapter 6, filtering the flights before grouping, as shown in Listing 7-6.

Listing 7-6. Rewriting the query without the view

```
SELECT
    bl.flight_id,
    departure_airport,
    (avg(price))::numeric (7,2) AS avg_price,
    count(DISTINCT passenger_id) AS num_passengers
FROM booking b
JOIN booking_leg bl USING (booking_id)
```

```
JOIN flight f USING (flight_id)
JOIN passenger p USING (booking_id)
WHERE actual_departure between '2023-08-01' AND '2023-08-14'
GROUP BY 1,2
```

The execution plan for this query is shown in Figure 7-4 and shows that the restrictions on the flight table are applied first. The execution time for this query is three minutes.

	QUERY PLAN
	text
1	GroupAggregate (cost=1773339.64..1874091.39 rows=3663700 width=30)
2	Group Key: bl.flight_id, f.departure_airport
3	-> Sort (cost=1773339.64..1782498.89 rows=3663700 width=18)
4	Sort Key: bl.flight_id, f.departure_airport
5	-> Hash Join (cost=654792.33..1223638.13 rows=3663700 width=18)
6	Hash Cond: (p.booking_id = b.booking_id)
7	-> Seq Scan on passenger p (cost=0.00..334860.49 rows=16318749 width=8)
8	-> Hash (cost=630294.22..630294.22 rows=1266969 width=26)
9	-> Hash Join (cost=390159.18..630294.22 rows=1266969 width=26)
10	Hash Cond: (b.booking_id = bl.booking_id)
11	-> Seq Scan on booking b (cost=0.00..145003.98 rows=5643298 width=14)
12	-> Hash (cost=368135.07..368135.07 rows=1266969 width=12)
13	-> Hash Join (cost=10657.51..368135.07 rows=1266969 width=12)
14	Hash Cond: (bl.flight_id = f.flight_id)
15	-> Seq Scan on booking_leg bl (cost=0.00..310506.66 rows=17893566 width=8)
16	-> Hash (cost=10052.84..10052.84 rows=48373 width=8)
17	-> Bitmap Heap Scan on flight f (cost=760.25..10052.84 rows=48373 width=8)
18	Recheck Cond: ((actual_departure >= '2023-08-01 00:00:00-05':timestamp with time zone) AND (actual_departure <= '2023-08-14 00:00:00-05':timestamp with time zone))
19	-> Bitmap Index Scan on flight_actual_departure (cost=0.00..748.15 rows=48373 width=0)
20	Index Cond: ((actual_departure >= '2023-08-01 00:00:00-05':timestamp with time zone) AND (actual_departure <= '2023-08-14 00:00:00-05':timestamp with time zone))

Figure 7-4. Execution plan for Listing 7-6

It is misleading when database textbooks, including those teaching PostgreSQL basics, state that views can be used “like tables.” In practice, views that were originally created solely to encapsulate a stand-alone query are often used in other queries, joined to other tables and views, including joining multiple times to tables already included in the view, without knowing what’s going on behind the scenes.

On the one hand, people usually create a view precisely for the purpose of encapsulation, so that others can use it without needing to figure out selection logic. On the other hand, this opacity is the cause of poor query performance. This effect becomes especially pronounced when some columns in a view are the results of transformation. Consider the view flight_departure in Listing 7-7.

Listing 7-7. View with column transformation

```

CREATE VIEW flight_departure as
SELECT
    bl.flight_id,
    departure_airport,
    coalesce(actual_departure, scheduled_departure)::date
    AS departure_date,
    count(DISTINCT passenger_id) AS num_passengers
FROM booking b
JOIN booking_leg bl USING (booking_id)
JOIN flight f USING (flight_id)
JOIN passenger p USING (booking_id)
GROUP BY 1,2,3

```

Executing the query

```

SELECT
    flight_id,
    num_passengers
FROM flight_departure
WHERE flight_id =22183

```

...the filter on the flight will be pushed inside the view, and the query will be executed in 63 ms. A user who isn't aware that `flight_departure` is a view might think that all columns have comparable performance and might be surprised to see the result when running the following query:

```

SELECT
    flight_id,
    num_passengers
FROM flight_departure
WHERE departure_date= '2023-08-01'

```

This query takes 2.5 sec or forty times longer to execute. The difference is because the `departure_date` column is a transformation, and as discussed in Chapter 5, no indexes can be utilized to apply this filter. The execution plan for this query is shown in Figure 7-5. Notice that although indexes on `scheduled_departure` and `actual_departure` are not used, PostgreSQL still correctly identifies the order of joins and searches the `flight` table first and then uses index access for tables `booking` and `passenger`.

	QUERY PLAN	
	text	
1	Subquery Scan on flight_departure (cost=512344.83..521400.17 rows=258724 width=12)	
2	-> GroupAggregate (cost=512344.83..518812.93 rows=258724 width=20)	
3	Group Key: bl.flight_id, f.departure_airport, ((COALESCE(f.actual_departure, f.scheduled_departure))::da...	
4	-> Sort (cost=512344.83..512991.64 rows=258724 width=16)	
5	Sort Key: bl.flight_id, f.departure_airport	
6	-> Nested Loop (cost=18858.24..484660.18 rows=258724 width=16)	
7	Join Filter: (b.booking_id = p.booking_id)	
8	-> Nested Loop (cost=18857.80..416906.41 rows=89471 width=36)	
9	-> Hash Join (cost=18857.37..376334.93 rows=89471 width=28)	
10	Hash Cond: (bl.flight_id = f.flight_id)	
11	-> Seq Scan on booking_leg bl (cost=0.00..310506.66 rows=17893566 width=8)	
12	-> Hash (cost=18814.67..18814.67 rows=3416 width=24)	
13	-> Seq Scan on flight f (cost=0.00..18814.67 rows=3416 width=24)	
14	Filter: ((COALESCE(actual_departure, scheduled_departure))::date = '2023-08-01'::d...	
15	-> Index Only Scan using booking_pkey on booking b (cost=0.43..0.45 rows=1 width=8)	
16	Index Cond: (booking_id = bl.booking_id)	
17	-> Index Scan using passenger_booking_id on passenger p (cost=0.43..0.64 rows=9 width=8)	
18	Index Cond: (booking_id = bl.booking_id)	

Figure 7-5. Execution plan when some indexes can’t be utilized

An even worse case of performance degradation is shown in Listing 7-8. Unfortunately, this is a real-life case. When a person using a view does not know what query was used to create it, they might use it to select data that is much easier to obtain from the underlying tables.

Listing 7-8. Selection of only one column from the view

```
SELECT
    flight_id
FROM flight_departure
WHERE departure_airport='ORD'
```

This query doesn't concern itself with the number of passengers on a flight; it merely selects flights departing from ORD for which any tickets were sold. And yet, the execution plan for Listing 7-8 is quite complex—see Figure 7-6.

	QUERY PLAN text
1	Subquery Scan on flight_departure (cost=886771.95..918054.44 rows=962538 width=4)
2	-> GroupAggregate (cost=886771.95..908429.06 rows=962538 width=20)
3	Group Key: bl.flight_id, f.departure_airport, ((COALESCE(f.actual_departure, f.scheduled_departure))::date)
4	-> Sort (cost=886771.95..889178.30 rows=962538 width=12)
5	Sort Key: bl.flight_id, ((COALESCE(f.actual_departure, f.scheduled_departure))::date)
6	-> Nested Loop (cost=9545.27..774662.60 rows=962538 width=12)
7	Join Filter: (b.booking_id = p.booking_id)
8	-> Nested Loop (cost=9544.84..520610.13 rows=334023 width=36)
9	-> Hash Join (cost=9544.41..367021.97 rows=334023 width=28)
10	Hash Cond: (bl.flight_id = f.flight_id)
11	-> Seq Scan on booking_leg bl (cost=0.00..310506.66 rows=17893566 width=8)
12	-> Hash (cost=9384.99..9384.99 rows=12753 width=24)
13	-> Bitmap Heap Scan on flight f (cost=243.26..9384.99 rows=12753 width=24)
14	Recheck Cond: (departure_airport = 'ORD'::bpchar)
15	-> Bitmap Index Scan on flight_departure_airport (cost=0.00..240.07 rows=12753 width=0)
16	Index Cond: (departure_airport = 'ORD'::bpchar)
17	-> Index Only Scan using booking_pkey on booking b (cost=0.43..0.46 rows=1 width=8)
18	Index Cond: (booking_id = bl.booking_id)
19	-> Index Only Scan using passenger_booking_id on passenger p (cost=0.43..0.64 rows=9 width=4)
20	Index Cond: (booking_id = bl.booking_id)

Figure 7-6. Execution plan for the query in Listing 7-8

This query runs for 3.1 seconds. However, a query that selects the same information without using the view

```
SELECT
    flight_id
FROM flight
WHERE departure_airport='ORD'
    AND flight_id IN (SELECT flight_id FROM booking_leg)
```

...will use available indexes and will run for just 120 ms.

Why Use Views?

Now that we've seen so many examples of the negative effects of using views, is there anything to say in their defense? Are there any situations in which views can improve query performance?

Internally in PostgreSQL, any creation of a view includes creation of *rules*, implicitly, in most cases. The `select` rules may restrict access to underlying tables. Rules, triggers, and automatic updates make views in PostgreSQL extremely sophisticated and provide functionality very similar to tables.

However, they do not provide any performance benefit. The best, and perhaps only justified, use of views is as a security layer or to define a reporting entity, to ensure that all joins and business logic are defined correctly.

Materialized Views

Most modern database systems allow users to create materialized views, but their implementations and precise behavior vary.

Let's begin with a definition.

A materialized view is a database object that combines both a query definition and a table to store the results of the query at the time it is run.

A materialized view is different from a view, because query results are stored, not just the view definition. This means that a materialized view reflects the data at the time it was last refreshed, not current data. It is different from a table, because you can't modify data in a materialized view directly, but you can only refresh it using a predefined query.

Creating and Using Materialized Views

Let's walk through an example to help illustrate the definition of a materialized view. Listing 7-9 creates a materialized view.

Listing 7-9. Create a materialized view

```

CREATE MATERIALIZED VIEW flight_departure_mv AS
SELECT
    bl.flight_id,
    departure_airport,
    coalesce(actual_departure, scheduled_departure)::date departure_date,
    count(DISTINCT passenger_id) AS num_passengers
FROM booking b
JOIN booking_leg bl USING (booking_id)
JOIN flight f USING (flight_id)
JOIN passenger p USING (booking_id)
GROUP BY 1,2,3

```

What happens when this command is run? First, in this particular case, it will take a very long time to execute. But when it finishes, there will be a new object in the database, which stores the results of this execution of the query in the database. In addition, the query itself will be stored along with the data. In contrast to views, when materialized views are referenced in queries, they behave exactly like tables. The optimizer won't substitute them with their defining queries, and they will be accessed as tables. Indexes can also be created on materialized views, although they cannot have primary and foreign keys:

```

CREATE UNIQUE INDEX flight_departure_flight_id
ON flight_departure_mv(flight_id);
--
CREATE INDEX flight_departure_dep_date
ON flight_departure_mv(departure_date);
--
CREATE INDEX flight_departure_dep_airport
ON flight_departure_mv(departure_airport);

```

Executing this query

```

SELECT
    flight_id,
    num_passengers
FROM flight_departure_mv
WHERE departure_date= '2023-08-01'

```

...will take only 60 ms, and the execution plan will show an index scan.

Refreshing Materialized Views

A REFRESH command populates the materialized view with the results of the base query at the time the refresh is executed. The syntax for the REFRESH command follows:

```
REFRESH MATERIALIZED VIEW flight_departure_mv
```

Materialized views in PostgreSQL are less mature than in some other DBMS, like Oracle. Materialized views cannot be updated incrementally, and the refresh schedule can't be specified in the materialized view definition. Each time the REFRESH command is executed, the underlying table is truncated, and the results of the SELECT statement are inserted. If an error occurs during refresh, the refresh process is rolled back, and the materialized view remains unchanged.

During refresh, the materialized view is locked, and its contents are unavailable to other processes. To make the prior version of a materialized view available during refresh, the CONCURRENTLY keyword is added:

```
REFRESH MATERIALIZED VIEW CONCURRENTLY flight_departure_mv
```

A materialized view can only be refreshed concurrently if it has a unique index. The concurrent refresh will take longer than regular refresh, but access to the materialized view won't be blocked.

Should I Create a Materialized View?

It is difficult to provide specific, universal conditions in which creating a materialized view is beneficial, but there follow some guidelines for decision-making. Since materialized view refreshes take time and selecting from a materialized view is going to be much faster than from a view, consider the following:

- How often does the data in the base tables change?
- How critical is it to have the most recent data?
- How often do we need to select this data (or rather how many reads per one refresh are expected)?
- How many different queries will use this data?

What should the thresholds for “often” and “many” be? It’s subjective, but let’s look at some examples to illustrate. Listing 7-10 defines a materialized view very similar to the view in Listing 7-9, except that it selects flights that departed yesterday.

Listing 7-10. Materialized view for yesterday’s flights

```
CREATE MATERIALIZED VIEW flight_departure_prev_day AS
SELECT
    b1.flight_id,
    departure_airport,
    coalesce(actual_departure, scheduled_departure)::date
    departure_date,
    count(DISTINCT passenger_id) AS num_passengers
FROM booking b
JOIN booking_leg b1 USING (booking_id)
JOIN flight f USING (flight_id)
JOIN passenger p USING (booking_id)
WHERE (actual_departure BETWEEN CURRENT_DATE -1
                                AND CURRENT_DATE
        )
    OR (
        actual_departure IS NULL
        AND scheduled_departure BETWEEN CURRENT_DATE -1
                                AND CURRENT_DATE
    )
GROUP BY 1,2,3
```

Information about flights that departed yesterday is not going to change, so it is safe to assume that the view will not need to be refreshed until the next day. On the other hand, this materialized view can be used in several different queries that will all perform faster if query results are materialized.

Let’s consider another potential candidate for materializing, Listing 6-29. Suppose a materialized view with the subquery is created, as in Listing 7-11.

Listing 7-11. Create a materialized view from the subquery

```

CREATE MATERIALIZED VIEW passenger_passport AS
SELECT
    cf.passenger_id,
    coalesce(max (custom_field_value )
              FILTER (WHERE custom_field_name = 'passport_num' ), '') AS
        passport_num,
    coalesce(max (custom_field_value )
              FILTER (WHERE custom_field_name = 'passport_exp_date' ), '') AS
        passport_exp_date,
    coalesce(max (custom_field_value )
              FILTER (WHERE custom_field_name = 'passport_country' ), '') AS
        passport_country
FROM custom_field cf
GROUP BY 1;

```

This materialized view is going to be very helpful. First, it has been shown already that this query takes a long time to execute, so time will be saved by pre-calculating the results. Second, passport information does not change (this information is associated with booking, and the same person will be assigned a different `passenger_id` in a different booking). It looks like a great candidate for a materialized view, if not for a few potential issues.

First, passengers are not required to submit their passport information during booking. This means that although once it is entered, this information will remain the same, for any particular flight, passport information may continue to be entered until the gate is closed. Hence, this materialized view will need to be constantly refreshed, and each refresh takes about ten minutes.

Second, this materialized view will keep growing. Unlike the previous example, when each day's refresh will cover data solely from the previous day, data about passengers' passports will grow, and it will take longer and longer to refresh a materialized view. Such situations are frequently overlooked in the early stage of a project, when there is little data in any table and materialized views refresh quickly. Since PostgreSQL does not allow an incremental materialized view refresh, a possible solution could be to create another table, with the same structure as the materialized view in Listing 7-11, and periodically upend new rows when new passport information becomes available.

However, if the latter solution is adopted, it's unclear why the `custom_field` table is needed in the first place, if data is needed in the format specified by the `passenger_passport` materialized view. This will be a topic in the next chapter, which discusses the impact of design on performance.

Do Materialized Views Need to Be Optimized?

Although the query of a materialized view is executed less frequently than the materialized view itself is used, we still need to pay attention to its execution time. Even when a materialized view is a short query (e.g., when it contains data for the previous day, as in Listing 7-9), it may end up doing full scans of large tables, if proper indexes are not in place or an execution plan is suboptimal.

As mentioned earlier, we don't accept the excuse that something doesn't need to be optimized because it runs infrequently, whether once a month, once a week, or once a day. No one is happy with reports that run for six hours, no matter how infrequently. In addition, these periodic reports are often all scheduled at the same time—usually 8 AM on a Monday—starting the week with more stress than anyone needs. The stress will be even more pronounced if some *dependencies* between materialized views are present. In this case, materialized views should be refreshed in a specific order, increasing the total wait time till the final report can be refreshed.

It's worth mentioning one more problem that can be caused by slow-refreshing materialized views. If we start to take a database backup after refresh of a materialized view started and finish it before the refresh ends, this materialized view will be stored in the backup in an inconsistent state, and if you restore from this backup (e.g., in your development environment), you won't be able either drop or recreate it. With everything being said, we believe it's evident that materialized views should be optimized in a manner like any other SQL query. The techniques discussed in Chapters 5 and 6 can and should be applied to materialized views.

Dependencies

When views and materialized views are created, a side effect is the creation of *dependencies*. Both views and materialized views have queries associated with them, and when any attribute involved in those queries is altered, the dependent views and materialized views need to be recreated.

Actually, PostgreSQL doesn't even permit an alter or drop on a table or materialized views if they have dependent views and materialized views. Making a change requires adding the CASCADE keyword to the ALTER or DROP command.

If views and materialized views are built on top of other views, altering or removing one column in one base table may result in recreating several dozen dependent database objects. Creating a view does not take substantial time, but rebuilding multiple dependent materialized views does, and all this time the materialized views will be unavailable, even if they allow concurrent refreshes.

Subsequent chapters discuss functions and stored procedures, which can eliminate such dependencies.

Partitioning

So far, this chapter has discussed different ways of splitting queries into smaller parts.

Partition is a different sort of division—dividing the data. A partitioned table consists of several partitions, each of which is defined as a table. The partitioned table itself is a virtual table and does not store any rows. Instead, each table row is stored in one of the partitions according to rules specified when the partitioned table is created.

Partition support is relatively new in PostgreSQL, and beginning with PG 10, improvements are made in every release, making partitioned tables easier to use. Currently, PostgreSQL supports the following partitioning methods: range, list, and hash partitioning.

The most common case is range partitioning, meaning that each partition contains rows that have values of an attribute in the range assigned to the partition. Ranges assigned to different partitions cannot intersect, and a row that does not fit into any partition cannot be inserted.

As an example, let's create a partitioned version of the `boarding_pass` table. The sequence of commands is shown in Listing 7-12.

Listing 7-12. Create a partitioned table

```
---create table
---
CREATE TABLE boarding_pass_part (
    boarding_pass_id SERIAL,
```

```

    passenger_id BIGINT NOT NULL,
    booking_leg_id BIGINT NOT NULL,
    seat TEXT,
    boarding_time TIMESTAMPTZ,
    precheck BOOLEAN,
    update_ts TIMESTAMPTZ
)
PARTITION BY RANGE (boarding_time);
--create partitions
--
CREATE TABLE boarding_pass_may
PARTITION OF boarding_pass_part
FOR VALUES
FROM ('2023-05-01'::timestamptz)
TO ('2023-06-01'::timestamptz) ;
--
CREATE TABLE boarding_pass_june
PARTITION OF boarding_pass_part
FOR VALUES
FROM ('2023-06-01'::timestamptz)
TO ('2023-07-01'::timestamptz);
--
CREATE TABLE boarding_pass_july
PARTITION OF boarding_pass_part
FOR VALUES
FROM ('2023-07-01'::timestamptz)
TO ('2023-08-01'::timestamptz);
--
CREATE TABLE boarding_pass_aug
PARTITION OF boarding_pass_part
FOR VALUES
FROM ('2023-08-01'::timestamptz)
TO ('2023-09-01'::timestamptz);
--
INSERT INTO boarding_pass_part SELECT * from boarding_pass;

```

Does Partitioning Improve Performance?

There is a common belief that partitioning of a large table improves query performance. This is not always the case. Moreover, partitioning can often decrease performance, and you may need to rewrite your old queries to make sure that performance won't be negatively affected.

The way partitioning *can* improve performance is that if a query contains conditions on the partitioning key, the search is limited to these partitions only. This makes partitioning especially useful for long queries where full scans are the best option: partitioning can significantly reduce the time for table scans. For index access the difference is negligible: partitioning removes only one level from the B-tree.

How should a partitioning key for a table be selected? The way the Postgres optimizer works, two conditions should be met: 1) the partitioning key should be used in (almost) all queries that run on this table, or at least in critical queries, 2) these values should be known prior to the SQL statement execution. The latter means that this value can't be passed as a parameter or as a result of a subselect.

Let's look at an example from Chapter 6, Listing 6-21. If this query is limited to boarding time between July 15 and July 31

```
SELECT
    city,
    date_trunc('month', scheduled_departure),
    sum(passengers) passengers
FROM airport a
JOIN flight f ON airport_code = departure_airport
JOIN (
    SELECT
        flight_id,
        count(*) passengers
    FROM booking_leg l
    JOIN boarding_pass b USING (booking_leg_id)
    WHERE boarding_time > '07-15-23'
    and boarding_time < '07-31-23'
    GROUP BY flight_id
) cnt
```



```

USING (flight_id)
GROUP BY 1,2
ORDER BY 3 DESC

```

...this will be still a long query, which will perform a full table scan of the boarding_pass table. The execution plan is identical to the one in Figure 6-18.

However, executing the similar query using the partitioned table `boarding_pass_part` (see Listing 7-13), this query will take advantage of partitions.

Listing 7-13. Querying a partitioned table

```

SELECT
    city,
    date_trunc('month', scheduled_departure),
    sum(passengers) passengers
FROM airport a
JOIN flight f ON airport_code = departure_airport
JOIN (
    SELECT
        flight_id,
        count(*) passengers
    FROM booking_leg l
    JOIN boarding_pass_part b USING (booking_leg_id)
    WHERE boarding_time > '07-15-23'
    and boarding_time < '07-31-23'
GROUP BY flight_id
) cnt
USING (flight_id)
GROUP BY 1,2
ORDER BY 3 DESC

```

The execution plan in Figure 7-7 proves that instead of scanning the whole table, the optimizer chooses to scan only one partition, since the query is filtered on boarding time. And while query runtime on a non-partitioned table is approximately the same regardless of filtering by boarding time, for a partitioned table, the execution time is more than twice as fast, because all the rows are located in one partition.

	QUERY PLAN text
1	Sort (cost=1526392.02..1526916.72 rows=209882 width=49)
2	Sort Key: (sum(cnt.passengers)) DESC
3	-> GroupAggregate (cost=1495417.22..1500664.27 rows=209882 width=49)
4	Group Key: a.city, (date_trunc('month':text, f.scheduled_departure))
5	-> Sort (cost=1495417.22..1495941.92 rows=209882 width=25)
6	Sort Key: a.city, (date_trunc('month':text, f.scheduled_departure))
7	-> Hash Join (cost=1412529.77..1471841.97 rows=209882 width=25)
8	Hash Cond: (f.departure_airport = a.airport_code)
9	-> Hash Join (cost=1412504.20..1470737.38 rows=209882 width=20)
10	Hash Cond: (cnt.flight_id = f.flight_id)
11	-> Subquery Scan on cnt (cost=1385229.70..1437525.94 rows=209882 width=12)
12	-> HashAggregate (cost=1385229.70..1435427.12 rows=209882 width=12)
13	Group Key: l.flight_id
14	Planned Partitions: 4
15	-> Hash Join (cost=604073.24..1038919.77 rows=6156621 width=4)
16	Hash Cond: (b.booking_leg_id = l.booking_leg_id)
17	-> Seq Scan on boarding_pass_july b (cost=0.00..300688.40 rows=6156621 width=8)
18	Filter: ((boarding_time > '2023-07-15 00:00:00-05':timestamp with time zone) AND (boarding_time < '2023-07-15 00:00:00-05':timestamp with time zone))
19	-> Hash (cost=310506.66..310506.66 rows=17893566 width=8)
20	-> Seq Scan on booking_leg l (cost=0.00..310506.66 rows=17893566 width=8)
21	-> Hash (cost=15398.78..15398.78 rows=683178 width=16)
22	-> Seq Scan on flight f (cost=0.00..15398.78 rows=683178 width=16)
23	-> Hash (cost=16.92..16.92 rows=692 width=13)
24	-> Seq Scan on airport a (cost=0.00..16.92 rows=692 width=13)

Figure 7-7. Execution plan with a partitioned table

Partitions may have their own indexes that obviously are smaller than an index on the whole partitioned table. This option might be beneficial for short queries. However, this might significantly improve performance only if almost all queries extract data from the same partition. The cost of search in a B-tree is proportional to its depth. An index on a partition, most likely, will eliminate only one level of the B-tree, while the choice of needed partition also requires some amount of resources. These resources are likely comparable with the amount needed for an extra index level. Of course, a query may refer to a partition instead of the whole partitioned table, hiding the cost of choosing the partition to the application issuing the query.

Therefore, the benefits of partitioning for short queries should not be overestimated.

Why Create a Partitioned Table?

After reading the previous section, readers may wonder whether there is any justification to create partitioned tables. In fact, there are many, and we cover them in this section.

Partitions can be added to a partitioned table or dropped. The `DROP` command is executed significantly faster than bulk `DELETE` and does not require subsequent vacuuming. A typical use case is a table partitioned on date ranges (e.g., partition per month), a new partition is added, and the oldest one is dropped at the end of every month.

In addition, it is possible to `ATTACH` and `DETACH` partitions. This way, you can add an existing table as a partition to a partition table and detach partitions without deleting data. For example, you can keep the last three months of data in a partitioned table, add a new partition in the beginning of each month, detach the oldest partition, and then attach it to the archiving table.

Partitioning may be used to distribute large amounts of data across several database servers: a partition can be a foreign table.

The primary reason for partitioning, however, is the maintenance. When a table reaches terabyte size, it becomes challenging to copy such table or to run `pg_dump` on it. Most importantly, it becomes difficult to run *autovacuum* on this table. We will cover *autovacuum* and its importance for maintaining a healthy PostgreSQL database; for now, we just want to mention that *autovacuum* is extremely unlikely to complete on a 1 TB table.

Parallelism

Starting in version 10, PostgreSQL has capability to enable parallel query execution. Two configuration parameters, `max_parallel_workers` and `max_parallel_workers_per_gather`, define how many parallel processes are available and how many can be spanned in one session.

We still recall the excitement of the community when this feature was introduced. Indeed, parallelism was and often is billed as a silver bullet to solve all performance problems, and we feel compelled to warn you about not setting expectations too high regarding parallelism—this is true in any RDBMS, not just PostgreSQL.

Parallel execution can be viewed as yet another way to split up the query: the amount of work needed to execute a query is divided between processing units (processors or cores).

Any parallel algorithm has a certain part that must be executed on a single unit. Also, additional overheads appear as a cost of synchronizations between parallel processes. For these reasons, parallel processing is mostly beneficial when bulk data is processed. Specifically, parallel execution is beneficial for massive scans and hash joins. Both scans and hash joins are typical for long queries, for which the speed-up is usually most significant.

In contrast, the speed-up for short queries is usually negligible. Parallel execution of different queries may improve throughput, but this is not related to parallel execution of a single query.

Sometimes the optimizer may replace index-based access (that would be used within sequential execution) with a parallel table scan. This may be caused by imprecise cost estimation. In such cases, parallel execution may be slower than sequential execution.

All execution plans in this book were created with parallelism turned off.

In addition, whatever scalability benefits are provided by parallel execution cannot fix poor design or compensate for inefficient code for a simple mathematical reason: scalability benefits from parallelism are at best linear, while the cost of nested loops is quadratic.

Summary

This chapter covered different ways to break queries into smaller functional parts and the advantages and disadvantages of each approach. It covered the potential pitfalls of one often-used optimization tool—temporary tables—and showed how common table expressions can be used as an alternative that doesn't stymie the query optimizer. It also discussed views and materialized views and their impact on performance. Finally, it briefly addressed partitioning and parallel execution.

CHAPTER 8

Optimizing Data Modification

Up to this point, our focus has been on optimizing *queries*, which means that only data retrieval has been covered. We haven't touched on anything related to data manipulation, that is, updating, removing, or adding records. This is the subject of this chapter, which discusses how data manipulation affects performance and how performance for these operations can be improved.

What Is DML?

Any database system has two languages: DDL (data definition language), used to create tables and other database objects, and DML (data manipulation language), which is used to query and modify data in the database. In PostgreSQL, both DDL and DML are parts of SQL, but some commands are related to DDL (`ALTER TABLE`, `CREATE MATERIALIZED VIEW`, `DROP INDEX`, etc.), while others are related to DML (`INSERT`, `UPDATE`, `DELETE`). It is also common to refer to these commands as DDL and DML, respectively, so a reference to “running DDL” means executing data definition commands and “running DML” means executing `INSERT`, `UPDATE`, or `DELETE`.

Two Ways to Optimize Data Modification

The execution of any DML statement consists of two parts: selecting the records to be modified and the data modification itself. In the case of `INSERT`, the first part may be omitted when constants are being inserted. However, if an `INSERT-SELECT` construct is used, the records that are needed for the insert must be found first.

For this reason, optimizing a DML statement consists of two parts: optimizing selection and optimizing data modification.

If the search part is the problem, then it is the `SELECT` part of the command that should be optimized. This is well covered in prior chapters. This chapter concerns itself with the second part—optimizing writing data.

In the overwhelming majority of cases, even OLTP systems execute significantly fewer DML statements than `SELECT` statements. This is the primary reason that people seldom talk about optimizing DML. However, long-running DML may cause problems not only because the updated data isn't available in the system in a timely manner but also because it can create *blocking locks*, which slow down the execution of other statements.

How Does DML Work?

To discuss the optimizations applicable to data modification SQL commands, we need to understand how they are executed.

Low-Level Input/Output

At the end of the day, any SQL operation, no matter how complex, comes down to a couple low-level operations: reading and writing individual database blocks. The reason is simple: data contained in the database can only be processed when blocks are fetched to main memory, and all modifications are first done in main memory and then written to disk.

A fundamental difference between reads and writes is that reads from disk must be completed before data can be processed; thus, a `SELECT` statement cannot be completed before all needed blocks are fetched into memory. In contrast, the changes inside a block are completed before the write starts; thus, a SQL operation can be completed without any delays. There is no need to wait until the modified data are actually written to the disk. This is somewhat counterintuitive: usually one would expect that an update requires more resources than a read.

Of course, writes do require much more resources than reads: the database must modify indexes and register updates in the WAL (write-ahead log). Still, this happens in main memory as far as single DML statements are concerned. WAL records are forced to disk asynchronously: by default, PostgreSQL guarantees that they are written at commit.

This sounds great: any INSERT, UPDATE, or DELETE appears to run much faster than a SELECT. If so, why are optimizations still needed?

There are two major reasons: First, writes are still needed and hence consume some amount of hardware resources, mostly I/O bandwidth. The cost of writes is tied to the cost of transferring pages and is not necessarily visible on any individual operation, but it still slows down processing and can even affect the performance of SELECT statements. An additional workload is produced in the background (e.g., modified blocks being written to disk) and during maintenance. Typically, this maintenance performs data restructuring, for example, the VACUUM operation in PostgreSQL. Some restructuring tasks block access to the modified object for the entire duration of the restructuring operation.

Second, modifications may interfere with other modifications and even with retrieval. As long as data is not modified, the order of processing is immaterial. Data can be accessed from different SELECT statements simultaneously. In contrast, modifications cannot be simultaneous, and the order of operations is crucial. In order to ensure correctness, some operations must be delayed or even declined. Correctness is the responsibility of the concurrency control (a.k.a. transaction processing) subsystem. Transaction processing is not the focus of this book; however, discussion of modifications cannot avoid some considerations related to the transactional behavior of the DBMS.

The Impact of Concurrency Control

To ensure the correct order of operations, transaction dispatchers usually rely on locking. If a transaction requires a lock and another transaction already has a conflicting lock, the execution is delayed until the conflicting lock is released. This is called lock waiting. Lock waiting is the primary cause of delays in modification operations.

Another function of concurrency control is to ensure that updates are not lost. Any updates performed by a committed transaction must be reliably stored on a hard drive before commit. The mechanism for this is the write-ahead log (WAL). All data modifications must be registered in WAL records on the hard drive before a transaction can commit. The WAL is written sequentially, and on slow rotating drives, sequential reads and writes are two orders of magnitude faster than random reads and writes. This difference is negligible on SSDs. Although there is no need to wait until all changes are written from the cache to the database, commits still must wait until the WAL is flushed.

As result, committing too frequently can significantly slow down processing. An extreme case is running each DML statement in a separate transaction. This actually happens if an application does not use transaction control statements and therefore the database wraps each statement into a separate transaction. On the other hand, transactions that are too long may cause slowness due to locking.

The preceding considerations apply to any high-performance database. Let’s look at techniques specific to PostgreSQL.

One of the distinguishing features of PostgreSQL is MVCC: multi-version concurrency control. PostgreSQL never performs “updates in place,” that is, never “updates” a record. Instead, a new version of an item (e.g., a table row) is created and stored in a free space in the same or a newly allocated block, while the previous version is still accessible.

Figure 8-1 shows the structure of a block from Figure 3-1 after a deletion (or update) of the second row. The space previously occupied by this row cannot be used for another row; and, in fact, the data is still accessible.

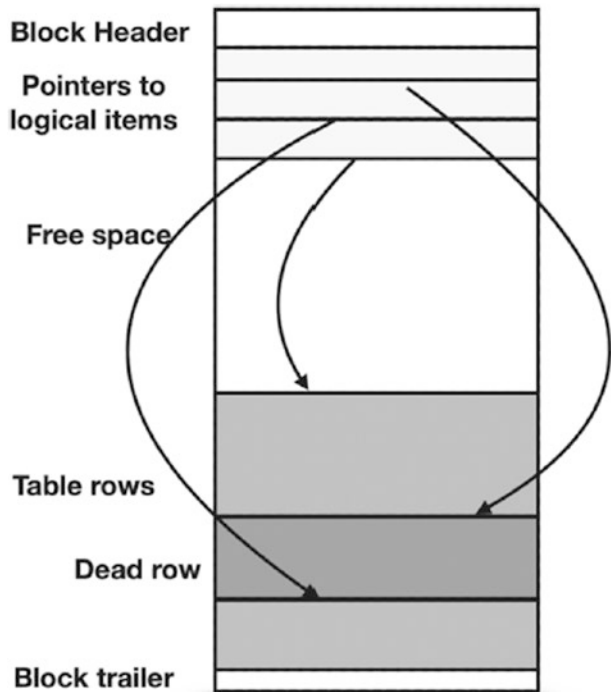


Figure 8-1. Block layout after deletion of a row

This feature may have both positive and negative impacts on performance.

The obsolete versions are not kept forever. A `VACUUM` operation removes them and consolidates the free space in a block when old versions are no longer needed for currently running transactions.

PostgreSQL uses the snapshot isolation (SI) concurrency control protocol to prevent undesirable interference between transactions. Note that database textbooks usually explain locking as it is used in a two-phase locking concurrency control protocol, which significantly differs from the way locks are used in PostgreSQL. Any intuition gained from textbooks or experiences with other systems might be misleading.

Under SI, a transaction always reads the latest committed version of a row. If another transaction has updated this row but did not commit before the start of the read operation, the read operation will return an obsolete version. This is an advantage as the older version is available and locking is not needed to read it. That is, the multi-version concurrency control improves throughput as there is no need to delay read operations.

According to SI, concurrent writes of the same data are not allowed: if two concurrent (i.e., running at the same time) transactions attempt to modify the same data, one of the two transactions must be aborted. In general, there are two strategies to enforce this rule. One is called *first update wins*, and the other is called *first commit wins*. It is easier to enforce the first strategy: we know that an update was executed right away, and the second transaction can be aborted without any wait. However, PostgreSQL utilizes the second strategy.

To enforce this rule, PostgreSQL uses write locks for any modification operation. Before a transaction can make any data changes, it has to acquire a lock for updates. If a lock cannot be obtained because some other transaction is modifying the same data, the operation is delayed until the termination of the transaction that holds the conflicting lock. If the lock is released because the holding transaction is aborted, the lock is granted to the waiting transaction, and a data modification operation can be completed. Otherwise, if the transaction commits successfully, the subsequent behavior depends on the transaction isolation level. For `READ COMMITTED`, which is the default in PostgreSQL, the waiting transaction will read the modified data, acquire a write lock, and complete the modification. This behavior is possible because on this isolation level, a read operation can read a version committed before the start of the `SELECT` statement, rather than the start of the transaction. If the isolation level is `REPEATABLE READ`, the waiting transaction will be aborted. This implementation results in waits but avoids unneeded aborts.

We do not discuss the `SERIALIZABLE` level because it is used vanishingly rarely.

Now, let's take a look at some important special cases.

Data Modification and Indexes

In Chapter 5, when we talked about creating new indexes, we mentioned that adding indexes to a table can potentially slow DML operations. How much slower depends on storage and system characteristics (e.g., disk speed, processors, and memory), but according to multiple PostgreSQL experts' observations, adding an extra index results in only a 1% increase in INSERT/UPDATE time.

You can perform some experiments using the `postgres_air` schema. For example, start with a table that has many indexes, such as the `flight` table.

First, create a copy of the table `flight` with no indexes:

```
CREATE TABLE flight_no_index AS
SELECT * FROM flight LIMIT 0;
```

Then, insert rows from the table `flight` into the table `flight_no_index`:

```
INSERT INTO flight_no_index
SELECT * FROM flight LIMIT 100
```

Note the execution time. Then, truncate the new table, and start building the same indexes built in Chapter 5 for the table `flight` on table `flight_no_index`. Repeat the insert. There is no difference in execution time for a small number of rows (around a couple hundred rows), but some slowness is observed when inserting 100,000 rows. For typical operations performed in the OLTP environment, however, there will be no material difference.

Naturally, creating indexes takes time, and it is worth mentioning that the `CREATE INDEX` operation in PostgreSQL locks the table for all DML operations (reads are still allowed). `CREATE INDEX CONCURRENTLY` takes longer to complete but allows DML to be executed.

As we mentioned earlier, PostgreSQL inserts new versions of updated rows. This has certain negative impacts on performance: in general, new versions are inserted into different locations, and therefore all indexes on the table must be modified. To reduce this negative effect, PostgreSQL uses a technique that sometimes is referred to as HOT (heap-only tuples); an attempt is made to insert the new version into the same block. If the block has sufficient free space and the update does not involve modifying any indexed columns, then there is no need to modify any indexes. In reality, things are a little bit more complicated, but we are not going to get into internals at this point.

DML and Vacuum

As mentioned earlier, PostgreSQL never removes old versions of rows immediately. The DELETE statement marks deleted rows as removed, while UPDATE inserts a new version of a row and marks the previous version as outdated. As soon as these rows are not needed for active transactions, they become dead. As the number of dead rows grows, it effectively reduces the number of active rows in a block and thus slows down subsequent heap scans.

The space occupied by dead rows (i.e., deleted tuples) remains unused until it is reclaimed by a VACUUM operation. This situation is called *table bloat*. Most of the time, even with a relatively high rate of updates, routine vacuuming initiated by the autovacuum daemon addresses the dead tuples promptly, so that they do not cause any significant delays.

The recommendations on tuning AUTOVACUUM system parameters can be found in PostgreSQL documentation and on many consulting companies' websites. While choosing the optimal values for these parameters, you will need to find a right balance between reducing I/O spikes, making sure vacuum can be completed relatively fast, and still having the job done (i.e., keeping bloat at bay).

Mass UPDATE/DELETE

What happens if a mass UPDATE/DELETE is performed, that is, an operation that affects a large fraction of the table? If you run this type of workload, you might notice that SELECTs from the affected table become significantly slower. The visibility map forces recheck to go to the heap blocks; in addition, the number of active tuples on a page decreases. This will result in more blocks needing to be read into memory for each select operation.

While there are multiple ways to tune autovacuum on a very busy system, if the level of updates/deletes is too high, it might be impossible to reduce table and index bloat, even with perfectly tuned autovacuum parameters. You might need to evaluate the whole system architecture to see whether the volume of data modification is justified. This includes checking whether applications perform several consecutive updates on the same set of records or whether it would make sense to partition a table and replace mass updates with DROP/DETACH partition and subsequently CREATE/ATTACH partition.

Frequent Updates

Now, let's consider a different case: a table experiences frequent updates (although each of these updates affects a single row or a very small number of rows).

Depending on the nature of these updates, sometimes it might be possible to reduce both the overhead of index updates and the table bloat by setting the table's `fillfactor`. This storage parameter sets the percentage of free space in table blocks, and it can be defined in the `WITH` clause of the `CREATE TABLE` statement. By default, the value of this parameter is 100, which tells PostgreSQL to fit as many rows as possible and minimize the size of free space in every block. Thus, usually free space can appear only after updates or deletions followed by vacuuming.

However, if the size of table rows is small enough to store multiple records in one block and updates do not affect indexed attributes, we can specify smaller values of `fillfactor`. PostgreSQL allows values as low as 10, leaving 90% of block space for updated versions of rows. Of course, small values of the `fillfactor` parameter result in an increased number of blocks needed to store table data and hence increase the number of reads needed for a heap scan of the table. This significantly slows down long queries but might be less significant for short queries, especially when only one row is selected from a block. Setting the `fillfactor` to lower than 100% increases the likelihood of HOT updates described earlier in this section.

Referential Integrity and Triggers

The presence of multiple foreign keys in a table can potentially slow DML. This is not to suggest that referential integrity checks are bad. On the contrary, the ability to maintain referential integrity is one of the most powerful features of relational systems. The reason they might slow data manipulation operations is that for each `INSERT/UPDATE` operation on a table with integrity constraints, the database engine has to check whether the new values of the constrained columns are present in the respective parent tables, thus executing additional implicit `SELECT` statements. These checks may take virtually no time, for example, if the parent table is a small lookup containing just a handful of rows. However, if the parent table size is comparable with the size of the child table, the overhead may be more noticeable. As in most other cases, the actual delay time depends on system parameters and hardware characteristics.

The execution times with and without constraints can be compared by creating a copy of the flight table:

```
CREATE TABLE flight_no_constr AS  
SELECT * FROM flight LIMIT 0;
```

Then, once again, start adding the same constraints as the flight table has to the flight_no_constr table, and once again, try to perform inserts. You may notice that adding an integrity check on the aircraft_code attribute does not impact the time to insert, but adding constraints on departure_airport and arrival_airport would noticeably slow down inserts.

Note that operations on the parent table are also affected: when a record in the parent table is updated or deleted, the database engine must check that there are no records in each of the child tables that reference the updated or deleted value.

Triggers can potentially slow down data modification operations as well, for the same reason as referential integrity constraints: each trigger invocation may result in executing multiple additional SQL commands. The extent to which each trigger slows down execution depends on its complexity.

It's worth noting that referential integrity constraints in PostgreSQL are implemented using system triggers, so all observations regarding integrity constraints are applicable to triggers. However, the potential performance impact of triggers does not mean that triggers should not be used. On the contrary, if there are some actions or checks that should be performed for any DML operation on the table, it is beneficial to implement them using database triggers instead of programming these checks in the application. The latter approach will be less efficient and would not cover the cases when the data in the table is modified directly in the database instead of accessing it through the application.

Summary

In this chapter, we briefly discussed the implication of data manipulation operations on system performance. Typically, DML commands are executed at least an order of magnitude less frequently than SELECT statements. However, if data modification inefficiencies are not addressed in a timely manner, they may result in blocking locks and affect the performance throughout the application.

CHAPTER 9

Design Matters

In the Introduction, we noted that optimization begins during requirements gathering and design. To be precise, it starts with system design, including the database design, but it is impossible to come up with the right design unless we invest time in gathering information about the objects that should be present in the database. In this chapter, we will discuss a variety of design options and will show how design decisions can impact performance.

Design Matters

Chapter 1 explained two different solutions for storing information about phone numbers, shown in Figures 1-1 and 1-2. Let's return to this example.

Listing 9-1 shows the table definitions used in the `postgres_air` schema. The `account` table contains information about user accounts, and the `phone` table contains information about all phone numbers associated with accounts. This relationship is supported by a foreign key constraint.

Listing 9-2 shows an alternative design, where all phones are stored together with account information.

Listing 9-1. Two-table design

```
/* account table */
CREATE TABLE account(
    account_id integer,
    login text,
    first_name text,
    last_name text,
    frequent_flyer_id integer,
    update_ts timestamp with time zone,
```

```

        CONSTRAINT account_pkey PRIMARY KEY (account_id),
        CONSTRAINT frequent_flyer_id_fk FOREIGN KEY (frequent_flyer_id)
        REFERENCES frequent_flyer (frequent_flyer_id)
    );
/*phone table */
CREATE TABLE phone(
    phone_id integer,
    account_id integer,
    phone text,
    phone_type text,
    primary_phone boolean,
    update_ts timestamp with time zone,
    CONSTRAINT phone_pkey PRIMARY KEY (phone_id),
    CONSTRAINT phone_account_id_fk FOREIGN KEY (account_id)
    REFERENCES account (account_id)
);

```

Listing 9-2. One-table design

```

/* account table */
CREATE TABLE account(
    account_id integer,
    login text,
    first_name text,
    last_name text,
    frequent_flyer_id integer,
    home_phone text,
    work_phone text,
    cell_phone text,
    primary_phone text,
    update_ts timestamp with time zone,
    CONSTRAINT account_pkey PRIMARY KEY (account_id),
    CONSTRAINT frequent_flyer_id_fk FOREIGN KEY (frequent_flyer_id)
    REFERENCES frequent_flyer (frequent_flyer_id)
);

```

There are multiple reasons the two-table design was chosen for the `postgres_air` schema; as discussed in Chapter 1, many people do not have landlines at home or a dedicated work phone. Many people have more than one cell phone or a virtual number, like Google Voice. All these scenarios can be supported with the two-table solution and can't fit into the one-table solution, unless we start to add columns to accommodate each of these cases. Indicating a primary phone in the one-table solution would require repeating one of the numbers in the `primary_phone` column, creating room for inconsistency. From a performance perspective, the two-table solution is also more beneficial.

In the two-table solution, searching for an account by phone number is a straightforward `SELECT` statement:

```
SELECT
    DISTINCT account_id
FROM phone
WHERE phone='8471234567'
```

This query will be executed using an index-only scan.

In the one-table design, a similar query would look like this:

```
SELECT
    account_id
FROM account
WHERE home_phone= '8471234567'
    OR work_phone= '8471234567'
    OR cell_phone= '8471234567'
```

To avoid a full scan, three different indexes must be built.

Does this mean that the one-table design is worse than the two-table design? It depends on how the data is accessed. If the schema is supporting a system used by travel agents, the most likely use case is needing to pull a customer account based on the phone number. When an agent asks a customer for their phone number, the customer is unlikely to specify the type of the phone.

On the other hand, consider a report on customer accounts that have been updated in the last 24 hours. This report should include home phone, work phone, and cell phone in separate columns regardless of if any are empty and should include accounts that have had any modification in the last 24 hours—including updates to phone numbers. In this case, the one-table solution, shown in Listing 9-3, is much simpler and more efficient.

Listing 9-3. Usage of one-table design

```
SELECT *
FROM account
WHERE update_ts BETWEEN now()- interval '1 day' AND now();
```

Producing the same result in the two-table design is more involved—see Listing 9-4.

Listing 9-4. Same query with a two-table design

```
SELECT
    a.account_id,
    login,
    first_name,
    last_name,
    frequent_flyer_id,
    home_phone
    work_phone,
    cell_phone,
    primary_phone
FROM account a
JOIN (
    SELECT
        account_id,
        max(phone) FILTER (WHERE phone_type='home')
            AS home_phone,
        max(phone) FILTER (WHERE phone_type='work')
            AS work_phone,
        max(phone) FILTER (WHERE phone_type='mobile')
            AS cell_phone,
        max(phone) FILTER (WHERE primary_phone IS true)
            AS primary_phone
    FROM phone
    WHERE account_id IN (
        SELECT
            account_id
```

```

FROM phone
WHERE update_ts BETWEEN now()- interval '1 day' AND now()
UNION
SELECT
    account_id
FROM account
WHERE update_ts BETWEEN now()- interval '1 day' AND now()
)
GROUP BY 1) p
USING (account_id);

```

These two examples are illustrative for another reason—the query for which the two-table solution is preferred is more likely to occur in an OLTP system, and the query that is better served by the one-table solution is more likely in an OLAP system. ETL tools can be used to transform data from OLTP systems to a format that is better suited to business intelligence needs.

A similar situation was shown in Chapter 6, where a nonoptimal database design prompted a nonoptimal query (see Listing 6-26). Even the optimized version of the query remained relatively slow. These examples illustrate impact of database design on performance and that sometimes the negative consequences of poor design can’t be remedied by improving the query or building additional indexes.

Subsequent sections of this chapter address the most common design choices that may negatively affect performance.

Why Use a Relational Model?

While all the previous examples are relational, as PostgreSQL is built on a relational model, we’re aware that many people think of relational databases as outdated or out of fashion. Public talks with titles like “What comes after relational databases?” are given at a regular cadence.

This section isn’t a defense of relational databases. Relational databases don’t need defending, and, so far, no would-be successor has had gained even close to a comparable level of adoption. Rather, the goal is to explain the limitations of other models.

Types of Databases

So what alternatives exist to relational models? There are a wide variety of database systems and data stores currently in use, using a wide variety of data models and storage techniques. These include traditional relational systems with row-based or column-based storage, scalable distributed systems, stream processing systems, and much more.

We've seen more than one non-relational database system work its way through the Gartner hype cycle, from the peak of overinflated expectations to the trough of disillusionment. It is worth noting, however, that the core of the relational model is a query language based on Boolean logic, rather than any specific way of storing data. This is likely the reason many systems created as alternatives to the traditional RDBMS ended up with variations of SQL as a high-level query language and hence the associated Boolean logic.

While relational databases do not appear close to being dethroned, there are technologies developed and validated within new systems that have proven useful and been widely adopted. Three popular approaches are entity-attribute-value models, key-value stores, and hierarchical systems, the last often referred to as document stores.

Entity-Attribute-Value Model

In the entity-attribute-value (EAV) model, values are scalar (often text, to accommodate multiple data types). To review from Chapter 6, this model features a table with three columns, the first for an identifier for an entity, the second an identifier for an attribute of that entity, and the third the value of that attribute for that entity. This is done in the name of “flexibility,” which in reality means imprecise or unknown requirements. Unsurprisingly, this flexibility comes at the expense of performance. Chapter 6 introduced the `custom_field` table, noted that this design was not optimal, and showed how it could negatively affect performance. Even after applying optimization techniques to avoid multiple table scans, execution was relatively slow.

In addition to performance impacts, this design curtails data quality management. In the case introduced in Chapter 6, three custom fields contain data of three different types: `passport_num` is a number, `passport_exp_date` is a date, and `passport_country` is a text field, which should contain a valid country name. However, in the `custom_field` table, they all are stored in the text field `custom_field_value`, which does not allow strong type checks or referential integrity constraints.

Key-Value Model

The key-value model stores complex objects within a single field, so the structure is not exposed to the database. Individual attributes of the object then are much more complex to extract, effectively kneecapping the database engine in doing tasks beyond returning a single object via a primary key. In the most extreme case, a design may package all fields other than the primary key into a single JSON object.

Since PostgreSQL introduced JSON support in version 9.2, this approach has become very popular with database and application developers. JSONB was introduced in version 9.4, and more enhancements have followed in every subsequent version. With this support, table columns defined as JSON are commonplace. For example, the passenger table from the `postgres_air` schema could be defined as shown in Listing 9-5.

Listing 9-5. Table with JSON

```
CREATE TABLE passenger_json (
    passenger_id INT,
    passenger_info JSON);
```

An example of the `passenger_info` JSON is shown in Listing 9-6.

Listing 9-6. Example of a JSON value

```
{"booking_ref" : "8HNB12",
"passenger_no": "1",
"first_name" : "MARIAM",
"last_name" : "WARREN",
"update_ts" : "2023-04-17T19:45:55.022782-05:00",
}
```

Yes, the suggested design looks universal and does not require any DDL changes no matter how many new data elements are added in the future. However, the same issues affect this design as the EAV model. This approach makes it impossible to perform type checks on scalar values, and referential integrity constraints can't be defined. Additionally, what is often overlooked is a substantial increase in the data volume required to store JSON. Indeed, the names of the keys are repeated in each record. Depending on the type and size of values stored, the data volume may increase several times over.

Tools and approaches for working with JSON fields are discussed later in this chapter.

Hierarchical Model

Hierarchical structures are easy to understand and use. In fact, hierarchical structures were first implemented in databases in the 1960s due to their ease of use, as well as their relatively small memory requirements. Of course, at that time, neither XML nor JSON was available. These structures work great as long as everything fits into a single hierarchy. However, using hierarchies becomes both complex and inefficient as soon as data fits into more than one hierarchy.

Let’s illustrate with examples from the `postgres_air` schema, shown in Figure 9-1. For an airport, the list of departing flights is one hierarchy, and the list of arriving flights is another. Boarding passes may fit into the same hierarchy as departing flights. At the same time, they can be a part of a completely different hierarchy that starts from bookings. Note that passengers and booking legs can’t fit into the same hierarchy without duplication.

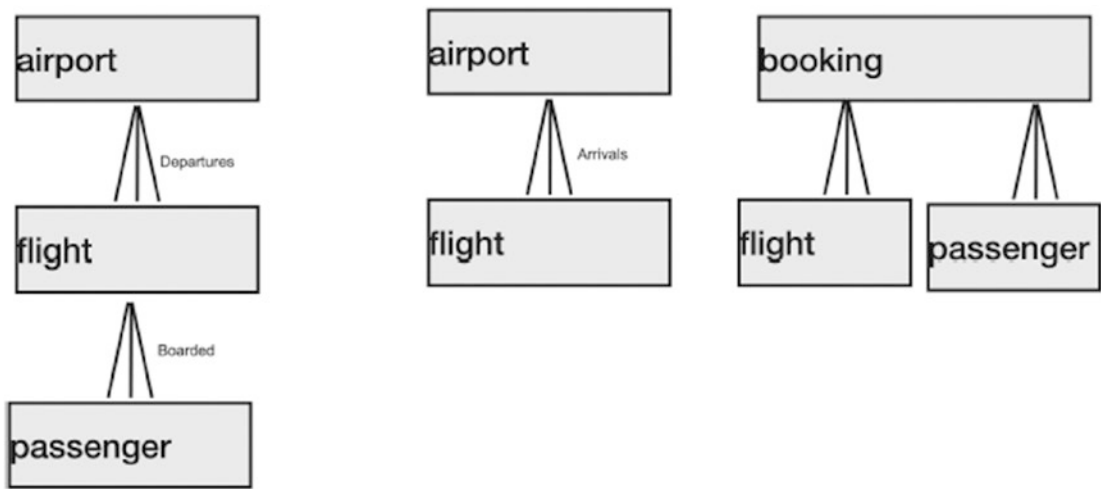


Figure 9-1. Examples of hierarchies in the `postgres_air` schema

The early hierarchical databases (IMS/360) provided several hierarchical views of data to the client application but supported more complex data structures internally.

Combining the Best of Different Worlds

PostgreSQL is not just a relational system. It is object-relational, meaning that column data types are not necessarily scalar. In fact, columns can store structured types including arrays, composite types, or objects represented as JSON or XML documents.

Using these features responsibly provides all the potential benefits of several alternative approaches in combination with more traditional relational features.

“Responsibly” is the operative word here. For example, PostgreSQL allows the multiple-hierarchy approach mentioned in the previous section. We can build hierarchical representations for a client application on top of an internal relational structure in the database. Such an approach combines the best of both worlds: data is extracted efficiently via the power of relational query, and the application can consume complex objects in a friendly data exchange format. More details on this approach are provided in Chapter 14.

Although distributed systems aren’t covered in this book, it’s worth mentioning that PostgreSQL has a massive set of extensions (additional libraries not included into basic distribution) that support distributed querying including DBMSs other than PostgreSQL. These extensions are called *foreign data wrappers (FDWs)*, and they provide almost transparent ways of accessing data that can reside in more than 60 types of DBMSs, both relational and non-relational.

Flexibility vs. Efficiency and Correctness

A frequent argument for a flexible design is that “the data structure/schema definition may change.” Then, the argument goes adding a column is a DDL (data definition) change and adding a row (in the key-value model) is just adding a row.

True, real-life systems evolve, and to adequately reflect these changes, existing data structures must be modified. This may entail adding or removing some attributes or changing data types or changing the cardinality of relationships. However, the inevitability of making changes does not necessitate the use of alternative models, such as document stores or key-value systems. The cost of making changes to the database schema must always be weighed against the potential performance and data integrity pitfalls of these flexible solutions.

The previous section mentioned the difficulty of creating any integrity constraints in a non-relational design. For some reason, there is a widespread belief that NoSQL databases are “faster” than relational databases. This statement may be true in a very limited number of scenarios, but in most cases, the situation is the opposite. There may be performance gains from horizontal distribution, but they are counterbalanced by the cost of needing to take additional steps to verify data integrity. Additional potential performance problems emerge because of the difficulties of creating indexes in EAV and key-value models.

For example, in the case of the `custom_field` table, `passport_exp_date` should be a date, and it is often compared with other dates, for example, with the date of the flight to make sure that the passport doesn’t expire prior to the date of departure. However, this date is stored in a text field, which means that it must be converted to a date to perform a type-specific compare. Moreover, this conversion can only be applied to rows that contain date-type values.

PostgreSQL has partial indexes, so it’s possible to create an index on only those rows that contain a passport expiration date. However, they cannot be indexed as a date that could be efficiently used as search criteria, because indexes cannot be built with mutable functions like so:

```
CREATE INDEX custom_field_exp_date_to_date
ON custom_field(to_date(custom_field_value, 'MM-DD-YYYY'))
WHERE custom_field_name='passport_exp_date'
```

This is because all date/time conversion functions are mutable, because they depend on the current session settings. In order to be able to use a conversion function in an index, one would need to write a custom conversion function. Chapter 11 covers creating user-defined functions. This function will have to include exception handling, which means that a value erroneously added in the wrong format won’t be indexed. In addition, the search itself will be significantly slower than having a date field in the first place.

What about the case of packaging all the attributes in a JSON column? Similar problems with indexing occur. It is possible to create an index on JSON; for example, for the table `passenger_json` in Listing 9-5, it is possible to create an index on `booking_ref` as shown in Listing 9-7.

Listing 9-7. Indexing a JSON column

```
CREATE INDEX passenger_j_booking_ref
ON passenger_json ((passenger_info ->> 'booking_ref'));
```

It will work slower than an index on the original passenger table, but it will work better than a sequential scan. However, for any value that is intended to be numeric or contain a date, the same conversion as the previous example will follow.

This is not to say there is no justification for any of these non-relational solutions.

For example, a table describing some regulations in the EU had about 500 columns and a single row added every time the regulations changed, approximately one row every five years. Replacement of this table with a variation of key-value (augmented with a couple of additional columns characterizing the value) made both database and application developers happy. There was no issue of efficiency because of the size of the data.

When considering JSON columns, our recommendation is to use them only in cases when the data only needs to be used as one whole object, such as when storing external documents (e.g., an externally sourced credit report). Even in these cases, if some attributes that will be used in search criteria can be isolated, it is advisable to parse them into additional, stand-alone columns as well as storing them as components of a larger object.

Must We Normalize?

There is hardly a term in relational theory more misused than “normalization.” It is commonplace for any number of DBAs, database developers, system architects, and others to declare that a system should be “normalized,” but few could provide a description of the outcome they are seeking, let alone a definition for normalization.

This isn’t just snobbery; it’s not necessary for everyone working in the field of data management to memorize the definitions of every normal form. Relational normalization algorithms aren’t often used in practice. In this sense, normalization is “dead” theory, the way that Latin is a dead language. However, scholars still find use in studying Latin, and analogously, some knowledge of normalization is essential for good database design.

Informally, a database schema is normalized if all column values depend only on the table primary key, and data is decomposed into multiple tables to avoid repetition.

One of the ways to create a normalized schema is to start by building an ER model: if entities are properly defined, the database schema produced from the ER model will be normalized. We can say that ER design implicitly includes discovery of dependencies. If it is not normalized, typically this indicates that some entities are missing.

Is it really important to normalize a database schema? Does it help performance? As is often the case, it depends.

The primary purpose of normalization is *not* to improve performance. Normalization creates a clean logical structure and helps ensure data integrity, especially when it is supported by referential integrity constraints. Normalization is needed for the same reasons that the relational model is needed: not necessarily to store the data, but to ensure data integrity and to be able to use relational querying language. The mapping between logical and storage structures is not necessarily one-to-one. Ideally, a clean logical structure should be provided for the application based on a storage structure optimized for performance.

On the other hand, there are many real-world entities that are denormalized and where normalization does not provide any benefits. The best-known example is the postal address. The US postal address consists of the following components: street address, city, zip code, and state.

It is not normalized, as anyone who has ever sent a package at an automated USPS kiosk knows. The automated check won't allow you to enter a zip code that does not match the previously entered address. However, we doubt that anyone would decide to normalize addresses when they are stored in a database table.

An often-heard argument in support of denormalized data structure is that “joins take time” and that denormalization is needed if we need queries to execute faster. Short queries, as discussed in Chapter 5, when constructed properly, require negligible additional time for joins, and this should not be traded for data accuracy.

However, there are cases when performance can indeed be improved through normalization, for example, when we need to select distinct values of some attribute with high selectivity or, in general, any subset of columns that are repetitive in a non-normalized table. In the `postgres_air` schema, the status of flights in the `flight` table is specified explicitly, which means that to find the list of possible flight statuses, one can execute the following query:

```
SELECT DISTINCT status
FROM flight
```

Use and Misuse of Surrogate Keys

Surrogate keys are unique values generated by the system in order to identify objects stored in the database. In PostgreSQL, surrogate values may be obtained as values selected from a sequence. When a row is inserted, a column specified with pseudo-type `serial` receives the next value from a sequence associated with the table automatically.

Surrogate keys are widely used. Some companies have internal standards that require use of surrogate keys for every table. However, surrogates have both advantages and disadvantages.

The advantage of surrogates is that the values assigned to different objects are guaranteed to be unique. However, the value of a surrogate is not related to other attributes of the object and is useless when a stored object needs to be matched to a real-world object.

Uniqueness of surrogates may hide certain errors. A real-world object may be represented in the database multiple times with different surrogate keys. For example, if a single purchase is registered in a system twice, a customer's card will be charged twice for a single product, and the issue is difficult to resolve without manual intervention. That being said, although our recommendation is to use some real-world unique attribute for a primary key, it is not always possible. In a database that stores all purchases in the supermarket, there is no way to distinguish between two bottles of Coke subsequently scanned by the same customer at the self-checkout. These two cases must be distinguishable in the source system: the purchase of two Coke bottles in one transaction, as opposed to a duplication of the same transaction to purchase a single Coke. Similarly, hospital systems may have multiple medical record numbers (MRNs) associated with one patient; it's critical to have a surrogate key in this case so all the patient's clinical data is stored together.

Sometimes, the presence of a surrogate key in the table is wrongly associated with normalization. A requirement to use surrogate keys is commonly explained as a way to make the database schema normalized. And, indeed, if there is a unique identifier assigned to each row, everything will be normalized. But since the unique identifiers would bear no relation to real-world objects, the result may be one real-world object mapped to multiple occurrences in the database. For example, we've seen a system in which each time a customer would enter their address, the system would assign a unique identifier for the city, if the city was not in the database yet. This way, the system ended up with six different versions of "Chicago." Needless to say, this has nothing to do with normalization and may jeopardize both data accuracy and performance.

The use of surrogate keys may also result in extra joins. The `flight` table refers to the `airport` table using three-character codes that are widely used in the real world. Airport codes can be extracted from the `flight` table in this design:

```
SELECT
    departure_airport,
    arrival_airport,
    scheduled_departure
FROM flight;
```

However, if a surrogate key was used for the `airport` table, returning airport codes would necessitate two trips to the `airport` table:

```
SELECT
    d.airport_code,
    a.airport_code,
    f.scheduled_departure
FROM flight f
JOIN airport d ON d.airport_id = f.departure_airport_id
JOIN airport a ON a.airport_id = f.arrival_airport_id
```

Let's take a closer look at the use of surrogate keys in the `postgres_air` schema.

The definition of the `airport` table in the `postgres_air` schema is shown in Listing 9-8. The primary key of that table is `airport_code`. This column contains three-character codes that are used to identify airports in all flight booking systems worldwide, and these codes never change. Hence, they are reliable as unique identifiers, and surrogate keys are not needed.

Listing 9-8. The `airport` table

```
CREATE TABLE airport(
    airport_code char(3)NOT NULL,
    airport_name text NOT NULL,
    city text COLLATE NOT NULL,
    airport_tz text NOT NULL,
    continent text,
    iso_country text,
    iso_region text,
```

```

        intnl boolean NOT NULL,
        update_ts timestamptz,
        CONSTRAINT airport_pkey PRIMARY KEY (airport_code)
    );

```

Similarly, aircrafts are identified with three-character codes, and we use these codes as a primary key for the aircraft table—see Listing 9-9.

Listing 9-9. The aircraft table

```

CREATE TABLE aircraft (
    model text,
    range numeric NOT NULL,
    class integer NOT NULL,
    velocity numeric NOT NULL,
    code text NOT NULL,
    CONSTRAINT aircraft_pkey PRIMARY KEY (code)
);

```

For the booking table (see Listing 9-10), the surrogate primary key `booking_id` is used, even though bookings have a six-character booking reference that uniquely identifies the booking and is never changed. The booking reference is also a surrogate key, although it is not derived from a database sequence. We could use the booking reference as a primary key. Thus, the column `booking_id` is redundant, though it provides some future-proofing if there is an anticipated need for bookings to come from more than one booking application. This also makes the table definition similar to definitions found in many industrial systems.

Listing 9-10. The booking table

```

CREATE TABLE booking (
    booking_id bigint NOT NULL,
    booking_ref text NOT NULL,
    booking_name text,
    account_id integer,
    email text NOT NULL,
    phone text NOT NULL,
    update_ts timestamptz,

```

```

    price numeric(7,2),
    CONSTRAINT booking_pkey PRIMARY KEY (booking_id),
    CONSTRAINT booking_booking_ref_key UNIQUE (booking_ref),
    CONSTRAINT booking_account_id_fk FOREIGN KEY (account_id)
    REFERENCES account (account_id)
);

```

The `booking_leg` table (Listing 9-11) links bookings to flights. Therefore, a natural key for this table would consist of `flight_id` and `booking_id`, that is, of two foreign keys referencing tables `flight` and `booking`. This pair of columns would be an excellent primary key. The decision to create an additional surrogate key `booking_leg_id` was driven by the idea to avoid references to the compound key from the dependent table (the `booking_leg` table is referenced from the `boarding_pass` table, which is the largest table in the database).

Listing 9-11. The `booking_leg` table

```

CREATE TABLE booking_leg (
    booking_leg_id SERIAL,
    booking_id integer NOT NULL,
    booking_ref text NOT NULL,
    flight_id integer NOT NULL,
    leg_num integer,
    is_returning boolean,
    update_ts timestamp with time zone,
    CONSTRAINT booking_leg_pkey PRIMARY KEY (booking_leg_id),
    CONSTRAINT booking_id_fk FOREIGN KEY (booking_id)
    REFERENCES booking (booking_id),
    CONSTRAINT flight_id_fk FOREIGN KEY (flight_id)
    REFERENCES flight (flight_id)
);

```

A surrogate key is needed for the `passenger` table (see Listing 9-12) because the same person may be a passenger on multiple bookings and a passenger is not necessarily registered as a client in the booking system (a flight may be booked by someone else on behalf of the passenger).

Listing 9-12. The passenger table

```

CREATE TABLE passenger(
    passenger_id serial,
    booking_id integer NOT NULL,
    booking_ref text,
    passenger_no integer,
    first_name text NOT NULL,
    last_name text NOT NULL,
    account_id integer,
    update_ts timestamptz,
    CONSTRAINT passenger_pkey PRIMARY KEY (passenger_id),
    CONSTRAINT pass_account_id_fk FOREIGN KEY (account_id)
    REFERENCES account (account_id),
    CONSTRAINT pass_booking_id_fk FOREIGN KEY (booking_id)
    REFERENCES booking (booking_id),
    CONSTRAINT pass_frequent_flyer_id_fk FOREIGN KEY (account_id)
    REFERENCES account (account_id)
);

```

There is no obvious way to identify accounts; therefore, a surrogate key must be used for the account table, shown in Listing 9-13.

Listing 9-13. The account table

```

CREATE TABLE account(
    account_id SERIAL,
    login text NOT NULL,
    first_name text NOT NULL,
    last_name text NOT NULL,
    frequent_flyer_id integer,
    update_ts timestamp with time zone,
    CONSTRAINT account_pkey PRIMARY KEY (account_id),
    CONSTRAINT frequent_flyer_id_fk FOREIGN KEY(frequent_flyer_id)
    REFERENCES frequent_flyer (frequent_flyer_id)
);

```

Loyalty program customers might be identified by card number. However, the separate surrogate key `frequent_flyer_id` facilitates replacements for a lost or stolen card, without needing to map or migrate data from one customer record to another or losing the data for a loyal customer.

In contrast, the surrogate `flight_id` in the `flight` table is needed. The natural identification of a flight consists of `flight_num` and `scheduled_departure`. The flight number is the same on different days, while departure time may vary on different days and can be changed slightly (e.g., 5–10 minutes later) when the flight is already partially booked. The `flight_id` represents a particular occurrence of the flight with a specific flight number as shown in Listing 9-14.

Listing 9-14. The flight table

```
CREATE TABLE flight (
    flight_id serial,
    flight_no text NOT NULL,
    scheduled_departure timestampz NOT NULL,
    scheduled_arrival timestampz NOT NULL,
    departure_airport char(3) NOT NULL,
    arrival_airport char(3) NOT NULL,
    status text NOT NULL,
    aircraft_code char(3) NOT NULL,
    actual_departure timestampz,
    actual_arrival timestampz,
    update_ts timestampz,
    CONSTRAINT flight_pkey PRIMARY KEY (flight_id),
    CONSTRAINT aircraft_code_fk FOREIGN KEY (aircraft_code)
    REFERENCES aircraft (code),
    CONSTRAINT arrival_airport_fk FOREIGN KEY (departure_airport)
    REFERENCES airport (airport_code),
    CONSTRAINT departure_airport_fk FOREIGN KEY (departure_airport)
    REFERENCES airport (airport_code)
);
```

The table `boarding_pass` (Listing 9-15) has a surrogate key, but it is not referenced from any other table and is therefore useless. The natural key of this table consists of two columns: `flight_id` and `passenger_id`.

Listing 9-15. Boarding pass

```
CREATE TABLE boarding_pass (  
    pass_id integer NOT NULL,  
    passenger_id bigint,  
    booking_leg_id bigint,  
    seat text,  
    boarding_time timestamptz,  
    precheck boolean,  
    update_ts timestamptz,  
    CONSTRAINT boarding_pass_pkey PRIMARY KEY (pass_id),  
    CONSTRAINT booking_leg_id_fk FOREIGN KEY (booking_leg_id)  
    REFERENCES booking_leg (booking_leg_id),  
    CONSTRAINT passenger_id_fk FOREIGN KEY (passenger_id)  
    REFERENCES passenger (passenger_id)  
);
```

Summary

This chapter discussed the impact of design decisions on performance. Discussion covered both choices within a relational model relating to normalization and surrogate keys, as well as popular non-relational models. The limitations of these models were explored, along with examples of alternative approaches.

CHAPTER 10

What About Configuration Parameters?

The Introduction to this book mentioned that most of the time, when people talk about “database tuning,” they assume that all performance problems can be fixed by figuring out optimal PostgreSQL configuration parameters.

Although some parameter tuning can indeed improve database performance, we are usually talking about 10%, 20%, and, in rare cases, up to 50% performance improvement, if parameters were previously set poorly. Those might sound like impressive numbers, but individual query optimization routinely makes queries run several times faster, in some cases, ten or more times faster, and restructuring applications (as we will see in Chapter 11) can improve overall system performance by hundreds of times.

This chapter discusses the most important PostgreSQL configuration parameters. We will demonstrate their impact on database performance and compare this impact with the techniques covered in previous chapters.

PostgreSQL Configuration Parameters Overview

There are over 300 PostgreSQL configuration parameters, and even an experienced DBA may not remember all of them. If you are an application developer or a database developer, you might not even know that PostgreSQL configuration parameters exist, let alone what they are and what they mean. There is a good reason for such ignorance since, in real life, ordinary users don’t have any say in how these parameters are set. Indeed, configuration parameters are set for the whole PostgreSQL instance, which may have multiple databases, so any individual user will get the same settings as every other user. To be completely transparent, in some cases, ordinary users **can** specify some parameters just for themselves, but let’s hold our horses for now.

Let's take a look at some configuration parameters that can make the most impact on the query performance.

Memory Allocation

There are a lot of myths related to PostgreSQL memory allocation parameters. Many recommendations and formulas, which can be found on the Internet, were issued many years and many PostgreSQL versions ago, and many of them have never been revisited since. As a result, many of these recommendations are either wrong or can be used only as a first approximation and tuned later.

shared_buffers

The data stored in a PostgreSQL database must be read from a hard drive to main memory before anyone can use it. The memory area allocated for this purpose is called shared memory buffers. This parameter sets the number of shared memory buffers used by the database server. The limit of 64 GB goes back to older versions of Linux and nowadays is mythical. This is one of the few memory-related parameters that PostgreSQL uses at its face value, meaning that it sets the hard limit for shared buffers. Also, resetting this parameter requires an instance restart. The only actual limitation to this value is that after subtracting it from total RAM, there should be enough memory left for `work_mem` (discussed later).

Recommendation: Start with 25% of RAM and increase it until `cache_hit_ratio` is close to 90%.

effective_cache_size

This parameter is advisory. It does not reserve any memory, and PostgreSQL uses it for query planning purposes only. It indicates how much memory the operating system has available for caching data (so it includes memory cache, disk cache, filesystem cache, etc.). The way the query planner uses it is not straightforward, but the higher it is defined, the higher the chance of using indexes rather than a sequential scan.

Recommendation: 75% RAM is usually recommended but not required (and this number is not related to the `shared_buffers` value).

Connections and Sessions

The next set of parameters is related to individual user sessions.

max_connections

This parameter sets the maximum number of concurrent connections to the PostgreSQL instance, which can be handled simultaneously. This is a hard limit; if the maximum number of connections is reached, the system will become non-responsive. No users except for a superuser will be allowed to connect. This situation will persist until the superuser kills some connections. Changing this parameter requires a system restart. You might wonder why this limit is necessary, and if it can't be changed on the fly, why not set it as high as possible so that we never run into these problems?

Too many connections are undesirable in any database because each connection takes up additional resources, even when idle. The most important resource is the memory allocated for each session—we will address the details in the next section. On the application side, this problem is addressed by setting up an application connection pool so that application users do not initiate new database connections but instead use one of the already opened connections from the available pool. If we set the value of *max_connections* really high, the system may become slow and non-responsive because there won't be enough resources for all connections. In addition, there are some internal limitations in the PostgreSQL kernel itself, which makes it undesirable to have more than several hundred connections at any given moment.

Now that we have described many of the problems that occur when there are too many connections, we should note that these horror stories are not guaranteed and are at least in part exaggerated. If there are enough hardware resources available on the host, even several thousand *max_connections* may work perfectly well.

Recommendation: Start from several hundred, observe system behavior, and adjust accordingly.

work_mem

The *work_mem* setting controls the amount of memory that can be used per query operation node and is generally used for sorting data and hash tables. This is also a hard limit, meaning that this is the maximum size of memory PostgreSQL will allocate to any query node.

Any hash operation that requires more than this amount of memory will resort to swapping to disk and will therefore take longer to complete.

A well-known formula suggests dividing 25% of the total system memory by *max_connections*. Therefore, a decrease in *max_connections* will allow for a higher value of *work_mem*. Note that in this case, like in many others, it is assumed that the

host has only one database, so be mindful when assigning the value for this parameter. Remember that PostgreSQL won't perform any calculations to check whether your settings make sense. Everyone might be happy until the next session cannot obtain enough RAM from the operating system and throws an out-of-memory error.

Fortunately, `work_mem` is one of the PostgreSQL parameters that can be set dynamically within a session or even for a duration of a transaction. So a possible solution could be to set the `work_mem` dynamically from the application. Suppose it is known that a particular query will be sorting a large dataset. In that case, it can be advantageous to increase the `work_mem` of that individual query (being careful not to over-allocate).

For example, if the session requires 200 GB for `work_mem`, you can run the following command:

```
set work_mem = '200GB';
```

This would set the session's `work_mem` to a new value and allow subsequent queries within that session to utilize more memory for sorting or hashing. To return to the standard value, run the following:

```
reset work_mem;  
maintenance_work_mem
```

This parameter determines the maximum amount of memory used for maintenance operations like `VACUUM`, `CREATE INDEX`, `ALTER TABLE`, `ADD FOREIGN KEY`, and data-loading operations (e.g., `COPY`). These operations may increase the I/O on database servers while they are running, so allocating more memory to them may lead to these operations finishing more quickly. There is a formula to calculate this value as well, but we will discuss this parameter in connection with the PostgreSQL `VACUUM` process.

Tuning Parameters for Better Performance

The statements like the one we made in the beginning of this chapter (comparing performance gains from parameter tuning to performance gains from query tuning) are notably difficult to illustrate. To follow along with the examples in this chapter, you will need to restore an additional clean copy of the `postgres_air` database. Alternatively, you can temporarily remove the indexes that we created in previous chapters and restore them after we are done with our experiments. Let's start with the query presented in Listing [10-1](#).

Listing 10-1. Query to optimize

```
SELECT
    f.flight_no,
    f.actual_departure,
    count(passenger_id) passengers
FROM flight f
JOIN booking_leg bl ON bl.flight_id = f.flight_id
JOIN passenger p ON p.booking_id=bl.booking_id
WHERE f.departure_airport = 'JFK'
    AND f.arrival_airport = 'ORD'
    AND f.actual_departure BETWEEN '2023-08-08' and '2023-08-12'
GROUP BY f.flight_id, f.actual_departure;
```

First, we will run it with default memory allocation parameters:

```
shared_buffers=128MB
work_mem=4MB
```

To observe the changes to buffer use, in this chapter, we use the EXPLAIN command with additional parameters—EXPLAIN (ANALYZE, BUFFERS, TIMING). The execution plan for this query is shown in Figure [10-1](#).

CHAPTER 10 WHAT ABOUT CONFIGURATION PARAMETERS?

QUERY PLAN
text
-> Gather Merge (cost=268857.90..268862.65 rows=10 width=24) (actual time=1070.505..1071.270 rows=11 loops=1)
Workers Planned: 2
Workers Launched: 2
Buffers: shared hit=923 read=171410
-> Partial GroupAggregate (cost=267857.88..267861.48 rows=5 width=24) (actual time=1064.207..1064.214 rows=4 loops=3)
Group Key: f.flight_id
Buffers: shared hit=923 read=171410
-> Sort (cost=267857.88..267859.06 rows=473 width=20) (actual time=1064.200..1064.204 rows=54 loops=3)
Sort Key: f.flight_id
Sort Method: quicksort Memory: 28kB
Buffers: shared hit=923 read=171410
Worker 0: Sort Method: quicksort Memory: 29kB
Worker 1: Sort Method: quicksort Memory: 28kB
-> Hash Join (cost=2683.38..267836.86 rows=473 width=20) (actual time=397.967..1064.161 rows=54 loops=3)
Hash Cond: (p.booking_id = bl.booking_id)
Buffers: shared hit=909 read=171410
-> Parallel Seq Scan on passenger p (cost=0.00..239655.37 rows=6798237 width=8) (actual time=0.329..834.198 rows=5437898 loops=3)
Buffers: shared hit=288 read=171385
-> Hash (cost=2681.74..2681.74 rows=131 width=20) (actual time=3.591..3.593 rows=69 loops=3)
Buckets: 1024 Batches: 1 Memory Usage: 12kB
Buffers: shared hit=599 read=25
-> Nested Loop (cost=268.50..2681.74 rows=131 width=20) (actual time=3.269..3.564 rows=69 loops=3)
Buffers: shared hit=599 read=25
-> Bitmap Heap Scan on flight f (cost=263.41..979.10 rows=5 width=16) (actual time=3.246..3.453 rows=4 loops=3)
Recheck Cond: ((departure_airport = 'JFK'::bpchar) AND (arrival_airport = 'ORD'::bpchar))
Filter: (actual_departure >= '2023-08-08 00:00:00-05':timestamp with time zone) AND (actual_departure <= '2023-08-12 00:00:00-05':timestamp with time zone)
Rows Removed by Filter: 178
Heap Blocks: exact=152
Buffers: shared hit=510 read=25
-> BitmapAnd (cost=263.41..263.41 rows=203 width=0) (actual time=3.174..3.175 rows=0 loops=3)
Buffers: shared hit=54 read=25
-> Bitmap Index Scan on flight_departure_airport (cost=0.00..112.77 rows=10179 width=0) (actual time=1.490..1.490 rows=10530 loops=3)
Index Cond: (departure_airport = 'JFK'::bpchar)
Buffers: shared hit=24 read=11
-> Bitmap Index Scan on flight_arrival_airport (cost=0.00..150.39 rows=13595 width=0) (actual time=1.420..1.420 rows=12922 loops=3)
Index Cond: (arrival_lairport = 'ORD'::bpchar)
Buffers: shared hit=30 read=14
-> Bitmap Heap Scan on booking_leg bl (cost=5.10..339.68 rows=85 width=8) (actual time=0.011..0.018 rows=17 loops=12)
Recheck Cond: (flight_id = f.flight_id)
Heap Blocks: exact=17
Buffers: shared hit=89
-> Bitmap Index Scan on leg_flight (cost=0.00..5.08 rows=85 width=0) (actual time=0.006..0.006 rows=17 loops=12)
Index Cond: (flight_id = f.flight_id)
Buffers: shared hit=38
Planning:
Buffers: shared hit=63 read=2
Planning Time: 2.154 ms
Execution Time: 1071.430 ms

Figure 10-1. Execution plan with default memory allocations

Looking at this query plan, we see that the optimizer uses two indexes on table *flight* and then scans all the blocks checking for the actual departure. Notice, also, that the number of shared buffers is insufficient (to achieve consistent results, run the same query or EXPLAIN ANALYZE more than once) and that PostgreSQL chooses to run two parallel workers to speed up the process. The total execution time is a just over one second.

Since we are not modeling the effect of multiple simultaneously running queries, the parameters that might make a material impact on the query execution time are `shared_buffers` (this change requires restart) and `work_mem` (can be modified in session). Let's first try to increase the size of `work_mem`, then change the `shared_buffers` parameter to 1 GB, restart the database, and repeat the `work_mem` changes.

Gradually increasing `work_mem` to up to 1 GB, we won't notice any significant changes in the execution plan and execution time. The limiting factor appears to be insufficient `shared_buffers`. We do not see any disk usage, which means that there is enough `work_mem` from the start. Figure 10-2 shows the execution plan with 128 MB `shared_buffers` and 500 MB `work_mem`. Execution time is fluctuating around 1 sec.

CHAPTER 10 WHAT ABOUT CONFIGURATION PARAMETERS?

QUERY PLAN	
text	
Finalize GroupAggregate (cost=268857.90..268862.75 rows=5 width=24) (actual time=980.568..981.901 rows=4 loops=1)	
Group Key: f.flight_id	
Buffers: shared hit=1236 read=171097	
-> Gather Merge (cost=268857.90..268862.65 rows=10 width=24) (actual time=980.559..981.894 rows=11 loops=1)	
Workers Planned: 2	
Workers Launched: 2	
Buffers: shared hit=1236 read=171097	
-> Partial GroupAggregate (cost=267857.88..267861.48 rows=5 width=24) (actual time=974.123..974.131 rows=4 loops=3)	
Group Key: f.flight_id	
Buffers: shared hit=1236 read=171097	
-> Sort (cost=267857.88..267859.06 rows=473 width=20) (actual time=974.111..974.115 rows=54 loops=3)	
Sort Key: f.flight_id	
Sort Method: quicksort Memory: 28kB	
Buffers: shared hit=1236 read=171097	
Worker 0: Sort Method: quicksort Memory: 30kB	
Worker 1: Sort Method: quicksort Memory: 27kB	
-> Hash Join (cost=2683.38..267836.86 rows=473 width=20) (actual time=408.195..974.076 rows=54 loops=3)	
Hash Cond: (p.booking_id = bl.booking_id)	
Buffers: shared hit=1222 read=171097	
-> Parallel Seq Scan on passenger p (cost=0.00..239655.37 rows=6798237 width=8) (actual time=0.591..739.594 rows=5437898 loops=3)	
Buffers: shared hit=576 read=171097	
-> Hash (cost=2681.74..2681.74 rows=131 width=20) (actual time=3.138..3.140 rows=69 loops=3)	
Buckets: 1024 Batches: 1 Memory Usage: 12kB	
Buffers: shared hit=624	
-> Nested Loop (cost=268.50..2681.74 rows=131 width=20) (actual time=2.716..3.098 rows=69 loops=3)	
Buffers: shared hit=624	
-> Bitmap Heap Scan on flight f (cost=263.41..979.10 rows=5 width=16) (actual time=2.689..2.978 rows=4 loops=3)	
Recheck Cond: ((departure_airport = 'JFK':bpchar) AND (arrival_airport = 'ORD':bpchar))	
Filter: ((actual_departure >= '2023-08-08 00:00:00-05':timestamp with time zone) AND (actual_departure <= '2023-08-12 00:00:00-05':timestamp with time ...	
Rows Removed by Filter: 178	
Heap Blocks: exact=152	
Buffers: shared hit=535	
-> BitmapAnd (cost=263.41..263.41 rows=203 width=0) (actual time=2.609..2.610 rows=0 loops=3)	
Buffers: shared hit=79	
-> Bitmap Index Scan on flight_departure_airport (cost=0.00..112.77 rows=10179 width=0) (actual time=1.157..1.157 rows=10530 loops=3)	
Index Cond: (departure_airport = 'JFK':bpchar)	
Buffers: shared hit=35	
-> Bitmap Index Scan on flight_arrival_airport (cost=0.00..150.39 rows=13595 width=0) (actual time=1.047..1.047 rows=12922 loops=3)	
Index Cond: (arrival_airport = 'ORD':bpchar)	
Buffers: shared hit=44	
-> Bitmap Heap Scan on booking_leg bl (cost=5.10..339.68 rows=85 width=8) (actual time=0.011..0.020 rows=17 loops=12)	
Recheck Cond: (flight_id = f.flight_id)	
Heap Blocks: exact=17	
Buffers: shared hit=89	
-> Bitmap Index Scan on leg_flight (cost=0.00..5.08 rows=85 width=0) (actual time=0.007..0.007 rows=17 loops=12)	
Index Cond: (flight_id = f.flight_id)	
Buffers: shared hit=38	
Planning:	

Figure 10-2. Execution plan with increased `work_mem` and default `shared_buffers`

Now, let's increase `shared_buffers` to 1 GB and restart our Postgres instance. Before we start measuring the execution time, we run this query a couple of times making sure that whatever can fit into the shared buffers is there.

Unfortunately, the execution time decrease is insignificant. The execution plan will stay the same, with slightly fewer reads due to the increase in `shared_buffers` (see Figure 10-3 for a portion of the execution plan).

Buffers: shared hit=1236 read=171097
-> Gather Merge (cost=268846.01..268850.76 rows=10 width=24) (actual time=707.388..708.271 rows=11 loops=1)
Workers Planned: 2
Workers Launched: 2
Buffers: shared hit=1236 read=171097
-> Partial GroupAggregate (cost=267845.99..267849.59 rows=5 width=24) (actual time=702.716..702.723 rows=4 loops=3)
Group Key: f.flight_id
Buffers: shared hit=1236 read=171097
-> Sort (cost=267845.99..267847.17 rows=473 width=20) (actual time=702.711..702.714 rows=54 loops=3)

Figure 10-3. A part of the execution plan with increased `shared_buffers`, which shows less reads than in Figure 10-2

If we keep increasing the `work_mem` to 200 MB, 500 MB, and 1 GB, we will notice a slow decrease in the execution time, finally getting as low as 750 ms. However, we need to be mindful that this increase in `work_mem` won't be possible on a production server for multiple sessions simultaneously. Anyway, this query does not seem to be too complicated. Are there other ways to improve its performance?

Are There Better Ways?

In all the execution plans produced during our experiments, there is one major deficiency: Postgres has to read the heap (table rows) in order to find records where the actual departure is between August 8 and August 12. That indicates that an index on this attribute might help. Let's go ahead and create the missing index:

```
CREATE INDEX flight_actual_departure
ON flight (actual_departure);
```

Immediately, there is a difference in the execution plan (see Figure 10-4).

Buffers: shared hit=284
-> Nested Loop (cost=524.84..2242.12 rows=131 width=20) (actual time=4.853..4.956 rows=69 loops=3)
Buffers: shared hit=284
-> Bitmap Heap Scan on flight f (cost=519.74..539.48 rows=5 width=16) (actual time=4.821..4.831 rows=4 loops=3)
Recheck Cond: ((departure_airport = 'JFK':bpchar) AND (arrival_airport = 'ORD':bpchar) AND (actual_departure >= '2023-08-08 00:00:00-05':timestamp with time zone) AND (ac
Heap Blocks: exact=4
Buffers: shared hit=195
-> BitmapAnd (cost=519.74..519.74 rows=5 width=0) (actual time=4.792..4.793 rows=0 loops=3)
Buffers: shared hit=183
-> Bitmap Index Scan on flight_departure_airport (cost=0.00..112.77 rows=10179 width=0) (actual time=1.451..1.451 rows=10530 loops=3)
Index Cond: (departure_airport = 'JFK':bpchar)
Buffers: shared hit=35
-> Bitmap Index Scan on flight_arrival_airport (cost=0.00..150.39 rows=13595 width=0) (actual time=1.351..1.351 rows=12922 loops=3)
Index Cond: (arrival_airport = 'ORD':bpchar)
Buffers: shared hit=44
-> Bitmap Index Scan on flight_actual_departure (cost=0.00..256.09 rows=16366 width=0) (actual time=1.579..1.579 rows=15873 loops=3)
Index Cond: ((actual_departure >= '2023-08-08 00:00:00-05':timestamp with time zone) AND (actual_departure <= '2023-08-12 00:00:00-05':timestamp with time zone))
Buffers: shared hit=104
-> Bitmap Heap Scan on booking_leg bl (cost=5.10..339.68 rows=85 width=8) (actual time=0.012..0.022 rows=17 loops=12)
Recheck Cond: (flight_id = f.flight_id)
Heap Blocks: exact=17
Buffers: shared hit=89
-> Bitmap Index Scan on leg_flight (cost=0.00..5.08 rows=85 width=0) (actual time=0.007..0.007 rows=17 loops=12)
Index Cond: (flight_id = f.flight_id)
Buffers: shared hit=38
Planning:
Buffers: shared hit=34
Planning Time: 0.430 ms
Execution Time: 540.307 ms

Figure 10-4. Execution plan with an index on the actual_departure column

In addition, the execution time finally decreased, and we are down to a 0.5 sec execution time. However, the difference is still not as dramatic as we would like it to be. Notice in examining the new execution plan that PostgreSQL performs a full scan on the passenger table (see Figure 10-5).

-> Hash Join (cost=2243.76..267385.35 rows=473 width=20) (actual time=490.297..2186.655 rows=54 loops=3)
Hash Cond: (p.booking_id = bl.booking_id)
Buffers: shared hit=498 read=171481
-> Parallel Seq Scan on passenger p (cost=0.00..239646.72 rows=6797372 width=8) (actual time=0.521..1941.157 rows=5437898 loops=3)
Buffers: shared hit=192 read=171481
-> Hash (cost=2242.12..2242.12 rows=131 width=20) (actual time=4.722..4.725 rows=69 loops=3)
Buckets: 1024 Batches: 1 Memory Usage: 12kB
Buffers: shared hit=284

Figure 10-5. A part of the execution plan with a full table scan

Noticing the join condition and foreign key constraint on the booking_id field, we might hypothesize that an index on booking_id might help. And, in fact, when a new index is created

```
CREATE INDEX IF NOT EXISTS passenger_booking_id
ON postgres_air.passenger
(booking_id);
```

...the execution plan changes dramatically—see Figure 10-6. Now, the total execution time is just 10 ms, **fifty times** less than with the best possible parameter tuning.

QUERY PLAN
text
HashAggregate (cost=2343.10..2343.15 rows=5 width=24) (actual time=7.839..7.843 rows=4 loops=1)
Group Key: f.flight_id
Batches: 1 Memory Usage: 24kB
Buffers: shared hit=360 read=41
-> Nested Loop (cost=525.28..2337.43 rows=1135 width=20) (actual time=5.715..7.765 rows=163 loops=1)
Buffers: shared hit=360 read=41
-> Nested Loop (cost=524.84..2242.12 rows=131 width=20) (actual time=5.634..5.838 rows=69 loops=1)
Buffers: shared hit=92
-> Bitmap Heap Scan on flight f (cost=519.74..539.48 rows=5 width=16) (actual time=5.588..5.623 rows=4 loops=1)
Recheck Cond: ((departure_airport = 'JFK':bpchar) AND (arrival_airport = 'ORD':bpchar) AND (actual_departure >= '2023-08-08 00:00:00-05':timestamp with time zone) AND (actual_departure <= '2023-08-12 00:00:00-05':timestamp with time zone))
Heap Blocks: exact=4
Buffers: shared hit=63
-> BitmapAnd (cost=519.74..519.74 rows=5 width=0) (actual time=5.550..5.552 rows=0 loops=1)
Buffers: shared hit=59
-> Bitmap Index Scan on flight_departure_airport (cost=0.00..112.77 rows=10179 width=0) (actual time=1.535..1.536 rows=10530 loops=1)
Index Cond: (departure_airport = 'JFK':bpchar)
Buffers: shared hit=11
-> Bitmap Index Scan on flight_arrival_airport (cost=0.00..150.39 rows=13595 width=0) (actual time=1.788..1.788 rows=12922 loops=1)
Index Cond: (arrival_airport = 'ORD':bpchar)
Buffers: shared hit=14
-> Bitmap Index Scan on flight_actual_departure (cost=0.00..256.09 rows=16366 width=0) (actual time=1.808..1.808 rows=15873 loops=1)
Index Cond: ((actual_departure >= '2023-08-08 00:00:00-05':timestamp with time zone) AND (actual_departure <= '2023-08-12 00:00:00-05':timestamp with time zone))
Buffers: shared hit=34
-> Bitmap Heap Scan on booking_leg bl (cost=5.10..339.68 rows=85 width=8) (actual time=0.022..0.040 rows=17 loops=4)
Recheck Cond: (flight_id = f.flight_id)
Heap Blocks: exact=17
Buffers: shared hit=29
-> Bitmap Index Scan on leg_flight (cost=0.00..5.08 rows=85 width=0) (actual time=0.015..0.015 rows=17 loops=4)
Index Cond: (flight_id = f.flight_id)
Buffers: shared hit=12
-> Index Scan using passenger_booking_id on passenger p (cost=0.43..0.64 rows=9 width=8) (actual time=0.024..0.027 rows=2 loops=69)
Index Cond: (booking_id = bl.booking_id)
Buffers: shared hit=268 read=41
Planning:
Buffers: shared hit=54 read=6
Planning Time: 1.913 ms
Execution Time: 8.061 ms

Figure 10-6. Execution plan with two indexes

More importantly, if we roll back the system parameters to the default settings and reduce the shared_buffers to 128 MB, nothing will change in either the execution plan or the execution time. No wonder—looking closely at the new execution plan in Figure 10-6, now that there are no massive table scans, the increased shared buffers are not needed anymore!

Other Limitations of Parameter Tuning

When we think about the meaning of the Postgres configuration parameters, it's important to remember that, in essence, all we are doing is communicating to the query planner what hardware resources it has available. Postgres is agnostic to the size of the RAM of the host on which it is running, the type of storage, how many cores are available, and what other systems are running on the same host. We provide some “input parameters” for optimization by setting up parameters to specific values.

For example, if we define `shared_buffers=128MB` when the host has 16 GB of RAM, the query planner won't be able to use most of the available memory. However, it could choose other ways to make queries faster, such as running them with multiple parallel processes, as we saw in Figure 10-3.

The opposite is also true. If we have the same 16 GB of RAM and we allocate 4 GB for shared buffers and at the same time set `default_work_mem=200MB` and `max_connections=700`, there is a high chance that Postgres will get an “out of memory” error from the operating system.

Another example: The `random_page_cost` parameter helps the optimizer estimate the cost of index-based access. The default value of 4 in older Postgres versions was a reflection of the characteristics of disks available ten years ago. Nowadays, keeping this value at 4 might stop the optimizer from choosing the best execution plan. However, if a system uses slower disks, reducing this parameter value would have the opposite effect.

Conclusion

Although it is important to set up PostgreSQL configuration parameters optimally for a specific system, it is far from enough to achieve the best performance. As we demonstrated, parameter tuning usually helps improve system performance by tens of percents, in some cases even more. At the same time, the creation of missing indexes can make queries run dozens of times faster and, at the same time, reduce resource usage.

CHAPTER 11

Application Development and Performance

Midway through this book, having covered multiple techniques along the way, it's time to step back and address the additional performance aspects that were foreshadowed in Chapter 1. We wrote then that the approach of this book is broader than just optimizing individual queries.

Database queries are parts of applications, and this chapter concerns optimizing processes rather than individual queries. Although this is not typically considered “database optimization” in its traditional sense, not addressing process deficiencies could easily cancel out any performance gains gleaned from optimizing individual queries. And since both application and database developers tend to ignore this area of potential improvement, we are going to claim it.

Response Time Matters

Chapter 1 enumerated reasons for poor performance, as well as covering why query optimization is necessary. What wasn't covered was *why* an application needs to be performant.

Hopefully, halfway through this book, you have not yet forgotten why you started reading it in the first place. Perhaps, the need to improve overall system performance or the performance of a specific part of the system became inescapably urgent. However, surprising as it may sound, it is still not uncommon to hear the opinion that slow response time is not such a big deal.

We reject this categorically: it *is* a big deal, and you do not need to go any further than your marketing department for confirmation. With today's consumer expectations, the saying *time is money* could not be more apt.

Multiple marketing research studies¹ have demonstrated that fast response time on a website or from a mobile app is critical to attract and maintain incoming traffic. In most cases, acceptable response time is below 1.5 seconds. If response time increases to over 3 seconds, 50% of visitors abandon a site, and more than three quarters of them never come back.

Specific examples include the numbers reported by Google that demonstrate that slowing search by 0.4 seconds results in a loss of eight million searches per day. Another example is Amazon finding that slowing page load time by one second results in \$1.6 billion of lost sales in a year. In cases like these, what problem must be addressed to improve the situation?

World Wide Wait

If you've ever talked to an application developer who works on a database application or if you are an application developer yourself, the following might sound familiar: "the application works perfectly fine until it hits the database!"

That statement, which we interpret as "an application often has performance issues when it interacts with a database," is often instead understood as "databases are slow," which is quite frustrating to hear. A DBMS is, after all, specialized software designed to provide *faster* data access, not to slow things down.

And in fact, if you ask the DBA overseeing database health on the same project, they might reply that the database is performing perfectly. If that's the case, why do users experience the "World Wide Wait"—interminable loading screens and slow response times?

Often, even if each individual database query executed by the application returns results in less than 0.1 seconds, the application page response time can be ten seconds or more. From the DBA's perspective, the system is performant, but the end user experience is poor. The problem, then, is not in the execution speed of each query but in the patterns of interactions between the application and the database.

¹www.fastcompany.com/1825005/how-one-second-could-cost-amazon-16-billion-sales/;
<https://builtvisible.com/improving-site-speed-talk-about-the-business-benefit/>

Performance Metrics

Chapter 1 discussed optimization goals, mentioning that many performance metrics, such as customer satisfaction, are “external” to the database and can’t be used by the optimizer. In fact, these metrics are external not only to the database but to the entire application.

The time needed to perform a given business function is difficult to measure and, consequently, difficult to improve. Chasing easily measured optimization goals can produce poor incentives. For example, an application developer can force a user to click ten buttons instead of one, and sometimes this will reduce the response time for each of the ten buttons. This might improve some benchmark results, but it would hardly improve user experience and satisfaction.

However, these are precisely the metrics the end user is interested in. They do not care about any individual query; they care about overall experience. They want an application to respond quickly, and they do not want to stare at “wait” and “loading” screens.

Impedance Mismatch

So what is the underlying cause of poor overall performance?

In very general terms, the cause is the incompatibility of database models and programming language models, often termed *impedance mismatch*. In electrical engineering, impedance is a generalization of the resistance to alternating current in a circuit when voltage is applied. The impedance phase angle for any component is the phase shift between the voltage across that component and current through that component; if this angle is close to 90 degrees, the delivered power is close to 0 even if both voltage and current are high.

Similarly, the expressiveness and efficiency of database query languages does not match the strengths of imperative programming languages. Even though both can have great strength, they might deliver less power than expected.

Both imperative programming languages and declarative query languages work extremely well to accomplish the tasks they were designed for. The problems start when we try to make them work together. The cause of poor performance is an incompatibility of database models and programming language models.

Applications and databases are designed to operate with

- Objects of different sizes (granularity)—single objects vs. bulk (sets of) objects
- Access patterns (navigation vs. search by attribute values)
- Different means of identification—address vs. a set of attribute values

In the remaining sections of this chapter, we discuss the consequences of this incompatibility in more detail.

A Road Paved with Good Intentions

The preceding might sound like we are blaming application developers for every performance problem and for their unwillingness to “think like a database.” But assigning blame is not a productive way to solve problems, including poor application performance. A more productive approach would be to try to understand how good intentions can lead to such excruciating results.

Let’s start by examining some common application development patterns.

Application Development Patterns

The most common modern software engineering architecture pattern is a layered architecture. Typically, there are four layers:

- End user interface
- Business logic
- Persistence
- Database

Each layer may only communicate with adjacent layers, and encapsulation and independence are encouraged both within each level and especially across levels. In this model, the business object “customer” is totally ignorant of the database table “Customer” and, in fact, could be connected to any arbitrary database so long as the persistence layer defined a mapping between the data in the database and the objects in the business layer.

There are a few important reasons for this, chief among them facilitating fast development, maintainability, ease of application updates, and making components reusable. It seems facially obvious that a change in the end user interfaces shouldn't de facto cause a change in the database schema. This strict separation also facilitates rapid work in parallel: developers can work on different parts of the application, and rest assured that the other parts of the application outside their narrow domains do not depend on the internal structure or implementation of the objects the developer is touching. And it of course seems useful that multiple applications can be built on the same foundation of business logic—that the internal logic of the application does not have to be duplicated for each new built environment.

So far, so good—so what is the problem? Unfortunately, there are many pitfalls, and the methodology doesn't quite deliver the promised benefit—at least, as it is implemented in the wild.

Consider the idea of centralizing business logic. First, the benefits of having all the logic in one place—the business layer—are somewhat reduced when that “one place” is several hundred thousand lines of code. In practice, such a large business layer will result in duplication—or, worse, attempted duplication. When the business logic layer is bloated, it's hard to find a function that does exactly what one wants—as a result, often, real-world business logic is implemented in different ways in different methods, with different results.

Second, this business logic may be available to additional end user interfaces, but it is not available to other business uses that interact directly with the database—most crucially, reporting. Thus, report writers end up duplicating application logic code, perhaps in a data warehouse or, worse yet, in individual reports, with no guarantee of equivalence with application logic.

Additionally, with this approach, communication with the persistence layer is typically limited to individual objects or even single scalar values, effectively disabling the power of the database engine. The end user interface might know all the different data elements that it needs, but because it doesn't communicate directly with the persistence layer, requests for data are mediated by the business logic layer.

A typical implementation of the persistence layer has data access classes that correspond one-to-one with business object classes. It is straightforward to write basic database DML functions (INSERT, UPDATE, DELETE), but what happens when operations must be performed on a set of objects of this class? There are two paths: The developer can create another set of methods that implement the same functions for a set of

objects. However, this would violate the principle of code reuse. Alternately, and more commonly, the developer simply iterates through the collection, calling the functions that have already been defined to handle each individual object in turn.

Imagine an application interface that lists all passengers departing from O'Hare airport. A database developer would assume that to list all passengers departing from O'Hare airport, they need to join the table `flight` with the table `boarding_pass`. All the information is returned in one go. For an application developer, the task might be trickier. They might have a method like `get_flight_by_departure_airport()` that takes an airport code as a parameter and returns a collection of flights. Then, they can iterate through the flights, returning all boarding passes for the flight. In effect, they are implementing a nested loop join algorithm inside the application.

To avoid this, they might use a few different solutions. They could add a departure airport attribute to the boarding pass object. However, this would open the door to data integrity problems: what if the flight departure time is updated in the flight record but not all boarding passes? Alternately, a method could be defined to retrieve boarding passes given a flight departure airport, but this would violate the precept of objects being ignorant of one another. In a pure layered approach, the boarding pass object is ignorant of the flight object, and the flight object is ignorant of the boarding pass. A method that pulls data for both wouldn't belong in either object.

“Shopping List Problem”

Stephane Faroult² describes the situation described previously as the “shopping list problem.”

Suppose you have a shopping list for the grocery store. In real life, you would go to the grocery store, get a cart, pick up all the items on your list, check out, pack the items up, go home, bring them inside, and put them into your fridge. Now imagine that instead, you go to the store, go inside, pick up just the first item from your shopping list, go back home, put the item in the fridge, and head to the store again! And you would continue in this fashion for every item on your list.

Does this sound ridiculous? Yes, but that's exactly what many applications do when it comes to their interaction with databases.

²Stephane Faroult and Peter Robson, *The Art of SQL*

Now imagine that in order to speed up the grocery run, experts suggested increasing the width of the aisles in the grocery store or building better roads to get there or devising a new sort of shopping cart.

Some of these suggestions could, indeed, help. But even if the shopping time was reduced by 30%, it would be a drop in the bucket compared to one simple process improvement: picking up all groceries in a single trip.

How can the shopping list problem be translated to application behavior? Most performance problems are caused by **too many queries that are too small**. And just as better highways can't improve the shopping experience if we continue to take an extra trip for each item on our list, the following popular suggestions do not help application performance:

- *More powerful computers* do not help much, as both the application and the database are in a wait state for 99% of time.
- *Higher network bandwidth* does not help either. High-bandwidth networks are efficient for transfer of bulk amounts of data but cannot significantly improve the time needed for roundtrips. Time depends on the number of hops and the number of messages but does not depend significantly on message size. Furthermore, the size of the packet header does not depend on the message size; hence, the fraction of bandwidth used for payload is small for very short messages.
- *Distributed servers* might improve throughput, but not response time, as the application sends requests for data sequentially.

The anti-pattern of “too many too small queries” has been a known problem for several decades. About 20 years ago, one of us had to analyze an application that needed five to seven minutes to generate an HTML form containing about 100 fields. The application code was perfectly structured into small, well-commented methods with nice formatting. However, the database trace showed that to produce this form, the application issued about 16,000 queries—more than the characters on the form being generated. Further analysis showed that a few thousand of the queries were coming from the method `GetObjectIdByName`. Each of these calls was followed by a query from the method `GetNameById` that was invoked from another part of the application,

probably written by another developer. The values of name were unique; therefore, the second call always returned the parameter of the first. A single query extracting all data needed to build the form returned the output in less than 200 milliseconds.

In the face of these predictable issues, many companies attempt many of the same, failed remedies to improve performance. Even if initially they are able to achieve some improvement, it does not last long. For example, we observed the optimization efforts of one company for several years.

Since the PostgreSQL optimizer is always trying to take advantage of available RAM, this company kept increasing hardware resources, making sure that the whole database could fit into main memory. They migrated from machines with 512 GB of RAM to 1 TB, 2 TB, and then 4 TB of main memory, when the only limiting factor was the availability of the respective configuration. Following each migration, after a short period of relative satisfaction, the problem would re-emerge: the database would grow bigger and stop fitting into main memory.

Another remedy that is often implemented is to use a key-value store instead of a full-featured DBMS. The argument usually goes something like “the application only uses primary keys to access data, so a query engine is not needed.” And, in fact, this approach may improve response time for a single data access. However, this cannot improve the time needed to complete a business function. In one extreme case, a record retrieval using a primary key value would take about ten milliseconds on average. At the same time, the number of database calls performed in one application controller action was nearly a thousand, with predictable overall performance impacts.

Interfaces

Yet another reason for suboptimal interactions between an application and a database is at the level of interfaces. Typically, applications use a generalized interface such as ODBC or JDBC. These interfaces provide an oversimplified view of a database as a set of flat tables. Actually, both the application and the database can operate in terms of complex structured objects; however, there is no way to transfer such high-level structures through the interface. Thus, an application cannot benefit from the high-level model even if it is maintained in the database.

To transfer a complex database object, an application is forced to use separate queries for each part of a database object or, alternately, to use a custom parsing method for deserializing the flat representation as returned over the interface into the complex objects themselves.

The imperfections of dominant development practices are well known to professionals. Why are these practices so common?

The reasons are not technical. Application developers almost always work under time pressure. A new product or a new feature has a release deadline, which is often “as soon as possible.” The financial gain of early delivery is significantly higher than that with later delivery and better quality.

Welcome to the World of ORM

The desire to isolate the database language (i.e., SQL) from application developers and thus simplify their task (and also reduce the need for the database skills) leads to the introduction of software that converts database functions into object methods.

An *object-relational mapper* (ORM) is software that maps a database object to an in-memory application object.

Some ORM developers have claimed that the impedance mismatch problem is solved. Objects are mapped one-to-one to database tables, and the underlying structure of the database, as well as the generated SQL used to interact with it, is of no concern to the application developer. Unfortunately, the cost of this solution is unacceptable performance degradation.

How does ORM work? The process is shown in Figure 11-1.

1. The application disassembles an object into undivisible (scalar) parts.
2. The parts are sent to/from the database separately.
3. In the database, the complex data structure is present, but all queries run separately.

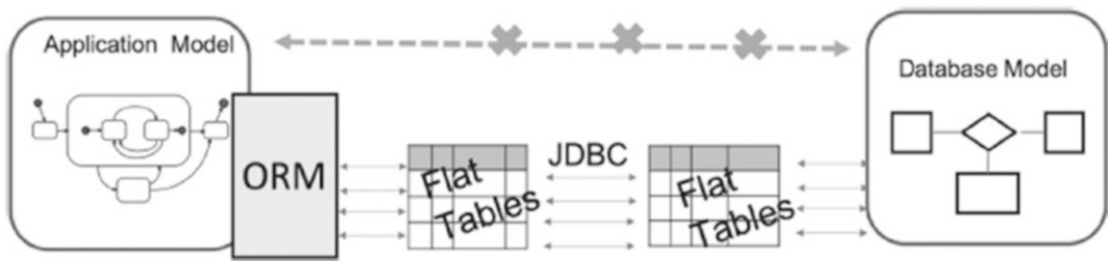


Figure 11-1. *How ORM works*

Theoretically, an ORM does not prevent the application from running arbitrary database queries; an ORM usually provides methods for that. However, in practice, generated queries are almost always used due to time pressures and the simplicity with which they are created in the application.

Because the actual database code is obscured from the developer, database operations on sets of objects end up happening very similarly to the non-ORM solution: an ORM method returns a list of object IDs from the database, and then each object is extracted from the database with a separate query (also generated in the ORM). Thus, to process N objects, an ORM issues $N + 1$ database queries, effectively implementing the shopping list pattern described in the previous section.

Such mapping solves the problem of abstraction from details of data storage but does not provide effective means to manipulate datasets.

In addition, the ORM might hide important implementation details. Take one example, observed in a production system: an `IsActive` flag on a `Customer` object to denote whether the customer had recent activity. A developer might think that this was just an attribute stored in the customer table of the database, but in actuality, it depended on a complex set of criteria based on customer behavior, and this query was run every time the attribute was invoked. Even worse, this attribute was used and frequently checked in the code for control flow and visual components that displayed differently based on the customer's status. As a result, this complex query ran multiple times to render a single page.

In Search of a Better Solution

To summarize the preceding, in the application layer, classes and methods for tables and sets should be integrated with the database to work effectively (methods should be executed by the DB engine). However, most architectures do not allow this kind of integration, which leads to reimplementations of database operations at the application layer.

This particular case of impedance mismatch is called *ORIM—object-relational impedance mismatch*.

Consequently, conventional ways of architecting communication between applications and databases are the most significant source of application slowness. There is no ill will here: application and database developers are doing the best they can with the tools they have.

To address this problem, we need to find a way to *transfer collections of complex objects*. Note that in fact, we are looking for a solution for two closely related problems. The first problem is inability to transfer “all the data at once,” that is, to think and operate in sets. The second problem is inability to transfer complex objects without disassembling them before the data transfer to and from the application.

In order to illustrate the desired outcome, let’s look at an example of how a web application may interact with the `postgres_air` database. When a user logs into the online booking system, the first thing they will see is most likely their existing bookings. When they select a specific booking, they will see a screen that looks something like the screenshot in Figure 11-2.

Your flight to Chicago (ORD) Jul 2 2023

Flight 3013 Munich (MUC) – New York (JFK) 4:10 PM

Flight 3014 New York (JFK) – Chicago (ORD) 12:10 PM – next day

Return flight – Jul 9 2023

Flight 3317 Chicago (ORD) – New York (LGA) 7:00 PM

Flight 3012 New York (JFK) – Munich (MUC) 5:05 AM – next day

Passengers

Passenger1: Kathleen Smith

Passenger 2: Doug White

Passenger 3; Martha White

Figure 11-2. *Your reservation screen*

The information that is displayed on your screen is drawn from several different tables: booking, booking_leg, flight, passenger, and airport. After check-in, you will also see the boarding passes.

A web application developed using a traditional approach would access the database 17 times to display these results: first, to select a booking_id list for the current user, then select booking details from the booking table, then select details for each booking leg (four in total), then select flight details for each flight (four more), and then select airport details (another four), and then passenger details (another three). However, an application developer knows exactly what objects to build to display booking results. On the database side, a database developer similarly knows how to select every piece of information needed to build this object. The structure of the object should look something like Figure 11-3.

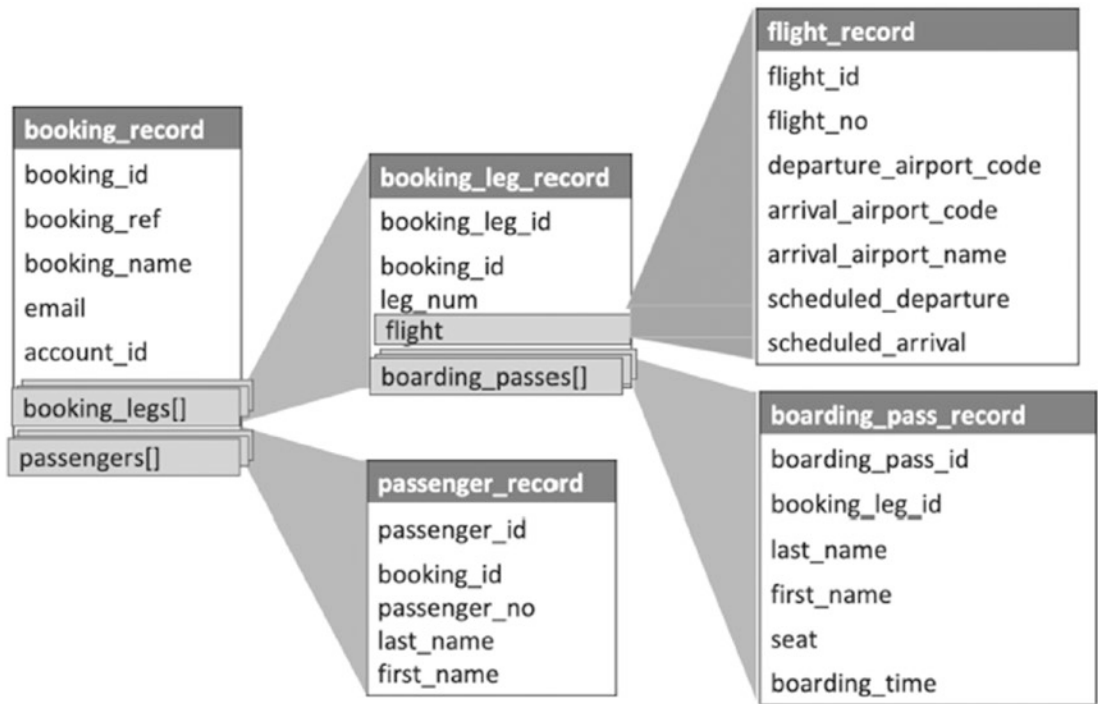


Figure 11-3. Mapping complex objects

If we could package data on the database side as such an object and send it to the application with one command, the number of database calls would decrease dramatically. And fortunately, PostgreSQL can build complex objects. The following PostgreSQL features make it possible:

- PostgreSQL is an object-relational database.
- PostgreSQL supports the creation of custom types.
- PostgreSQL functions can return sets, including sets of records.

Subsequent chapters will discuss functions that return sets of records and support of JSON/JSONB data types and custom data types and will show examples of how to create these functions and use them in applications.

Summary

This chapter discussed additional performance aspects that typically are not considered related to database optimization. Although, technically speaking, it is not about optimizing queries, it presents an approach to optimizing overall application performance. As we've often stated, SQL queries are not executed in a vacuum; they are parts of an application, and the "in between" area of communication between the application and database was and still is often omitted from consideration by both database and application developers.

For this reason, we've taken the liberty of claiming ownership of this uncharted territory and suggesting a path for improvement. Notably, this chapter does not provide any practical solutions or any examples of "how to do it right." Subsequent chapters discuss several techniques that together create a powerful mechanism to overcome the limitations of traditional ORMs.

CHAPTER 12

Functions

This chapter focuses on the most underused and misused features of PostgreSQL, objects—functions. Since all modern programming languages include user-defined functions, people often assume that database functions are cut from the same cloth and if you know how and when to write functions in an application programming language, you can apply this directly to PostgreSQL. This could not be further from truth.

This chapter discusses how PostgreSQL functions are different from functions in other programming languages, when functions should be created and when they should not, how using functions can improve performance, and how it can lead to a major performance degradation.

Before proceeding, let's address the widespread belief that using functions decreases portability. This is true as it goes, but consider the following:

- Neither SQL statements nor ORMs are 100% portable; some (hopefully minor) work will always be required.
- Swapping databases for an existing production system is always a major project and is never done on the fly. Some changes to the application itself are unavoidable. Converting functions adds relatively small overhead to the project.

Function Creation

PostgreSQL has both built-in (internal) functions and user-defined functions. In this respect, it is not any different from other programming languages.

Internal Functions

Internal functions are written in the C language and are integrated with the PostgreSQL server. For each data type supported by PostgreSQL, there are a number of functions that perform different operations with variables or column values of that type. Similar to imperative languages, there are functions for mathematical operations, functions to operate on strings, functions to operate on date/time, and many others. The list of available functions and supported types expands with each new release.

Some examples of built-in functions are shown in Listing 12-1.

Listing 12-1. Examples of built-in functions

```
sin(x);
substr(first_name,1,1);
now();
```

User-Defined Functions

User-defined functions are functions created by users. PostgreSQL supports three kinds of user-defined functions:

- Query language functions, that is, functions written in SQL
- C functions (written in C or C-like languages, like C++)
- Procedural language functions, written in a supported procedural language (referred to as PL)

A CREATE FUNCTION command is shown in Listing 12-2.

Listing 12-2. CREATE FUNCTION command

```
CREATE FUNCTION function_name (par_name1 par_type1, ...)
RETURNS return_type
AS
<function body>
LANGUAGE function_language;
```

From a PostgreSQL perspective, the database engine captures only the function signature—the name of the function and the list of parameters (which may be empty), the type of return value (which may be void), and some specifications, such as the

language in which it is written. For all procedural languages except for SQL, the function body is packaged into a string literal, which is passed to a special handler that knows the details of the language. The handler can either do all the work of parsing, syntax analysis, execution, and so on itself, or it can serve as the “glue” between PostgreSQL and an existing implementation of a programming language. The standard PostgreSQL distribution supports four procedural languages: PL/pgSQL, PL/Tcl, PL/Perl, and PL/Python. In this book, we will discuss only functions written in PL/pgSQL.

Introducing Procedural Language

Since we’re covering functions, it seems like a good idea to formally introduce the language (or languages) in which these functions can be written.

So far in this book, the only language that has been used is SQL, and the only operator used in code snippets, other than CREATE operators, was SELECT. Now, it is time to introduce procedural languages. In this book, we discuss only the PostgreSQL native procedural language PL/pgSQL.

A function written in PL/pgSQL can include any SQL operators (potentially with some modifications) and control structures (IF THEN ELSE, CASE, LOOP) and calls to other functions.

Listing 12-3 shows an example of a function written in PL/pgSQL that converts a text string to numeric, if possible, and returns null if the string does not represent a number.

Listing 12-3. Function converting text to numeric

```
CREATE OR REPLACE FUNCTION text_to_numeric(input_text text)
RETURNS numeric AS
$BODY$
BEGIN
    RETURN replace(input_text, ',', '::numeric;
EXCEPTION WHEN OTHERS THEN
    RETURN NULL::numeric;
END;
$BODY$
LANGUAGE plpgsql;
```

Using the information in Listing 12-2, we can identify the parts common to all user-defined functions. The function name is `text_to_numeric`, and it has only one parameter, `input_text`, of type `text`.

The RETURNS clause defines the type of value the function returns (numeric), and the LANGUAGE clause specifies the language in which the function is written, plpgsql.

Now, let's take a closer look at the function body.

Dollar Quoting

In the previous section, we stated that a function body is represented as a string literal; however, instead of quotes, it starts and ends with \$BODY\$. This notation in PostgreSQL is called dollar quoting, and it is especially useful when you create a function. Indeed, if you have a relatively large text of a function, chances are you will need to use single quotes or backslashes, and then you will need to double them at each occurrence. With dollar quoting, you can use two dollar signs, possibly with some tag between them to define a string literal.

Tagging makes this way of defining string constants especially convenient, because you can nest strings with different tags. For example, the beginning of a function body can look like Listing 12-4.

Listing 12-4. The usage of nested dollar quoting

```
$function$
DECLARE
V_error_message text:='Error: ';
V_record_id integer;
BEGIN
...
v_error_message:=v_error_message||$em$Record can't be updated,
#$em$||quote_literal(v_record_id);
...
END;
$function$
```

Here, we use dollar quoting with the tag function for a function body. Note that there are no rules limiting which tags can be used for the function body, including the empty tag; the only requirement is that the string literal ends with the same tag as the tag that started it. The examples here will use different tags as a reminder that tags are not predefined.

The function body starts with a `DECLARE` clause, which is optional and can be excluded if no variables are needed. `BEGIN` (without a semicolon) denotes the start of the statements section, and the `END` keyword should be the last statement in the function body.

Please note this section is not a comprehensive guide to function creation. Please refer to PostgreSQL documentation for more details.

More details are discussed as they are introduced in future examples.

Function Parameters and Function Output: Void Functions

Most often, a function will have one or more parameters, but it might have none. For example, the internal function `now()`, which returns the current timestamp, does not have any parameters. We can assign default values to any function parameter, to be used if no specific value is explicitly passed.

In addition, there are multiple ways to define function parameters. In the example in Listing 12-3, parameters are named, but they can also be positioned (`$1`, `$2`, etc.). Some parameters may be defined as `OUT` or `INOUT`, instead of specifying a return type. Again, this chapter doesn't intend to cover every possible specification, because function performance does not depend on all these specification variations.

Finally, it's important to note that a function may return no value; in this case, a function is specified as `RETURNS VOID`. This option exists because previously, PostgreSQL did not have support for stored procedures, so the only way to package multiple statements together was inside a function.

Function Overloading

Similar to other programming languages, functions in PostgreSQL can be polymorphic, that is, multiple functions can use the same name with a different signature. This feature is called *function overloading*. As mentioned earlier, a function is defined by its name and its input parameter set; the return type may be different for different sets of input parameters, but for obvious reasons, two functions cannot share both the same name and same set of input parameters.

Let's take a look at the examples in Listing 12-5. In case #1, a function that calculates the number of passengers on a specific flight is created. In case #2, a function with the same name, which calculates the number of passengers departing on a specific date from a specific airport, is created.

However, if you try to run snippet #3 to create a function that calculates the number of passengers on a specific flight number on a specific date, you will get an error:

ERROR: cannot change name of input parameter "p_airport_code".

You can create a function with the same name and with a different set of parameters and a different return type; thereby, you can create another function with the same name, as shown in case #4. However, if you try to create a function with the same name and with a different return type but the same parameters (case #5), the database will throw an error:

ERROR: cannot change return type of existing function

Listing 12-5. Function overloading

#1

```
CREATE OR REPLACE FUNCTION num_passengers(p_flight_id int) RETURNS integer;
```

#2

```
CREATE OR REPLACE FUNCTION num_passengers(p_airport_code text, p_departure
date) RETURNS integer;
```

#3

```
CREATE OR REPLACE FUNCTION num_passengers(p_flight_no text, p_departure
date) RETURNS integer;
```

#4

```
CREATE OR REPLACE FUNCTION num_passengers(p_flight_no text) RETURNS
numeric;
```

#5

```
CREATE OR REPLACE FUNCTION num_passengers(p_flight_id int) RETURNS numeric;
```

Note that the source code of these functions differs significantly. Listing 12-6 shows the source code of the `num_passengers(integer)` function, and Listing 12-7 shows the code of the `num_passengers(text, date)` function.

Listing 12-6. Source code of num_passengers(int)

```

CREATE OR REPLACE FUNCTION num_passengers(p_flight_id int) RETURNS integer
AS
$$BEGIN
RETURN (
    SELECT count(*)
    FROM booking_leg bl
    JOIN booking b USING (booking_id)
    JOIN passenger p USING (booking_id)
    WHERE flight_id=p_flight_id);
END;
$$ LANGUAGE plpgsql;

```

Listing 12-7. Source code of num_passengers(text, date)

```

CREATE OR REPLACE FUNCTION num_passengers(
    p_airport_code text,
    p_departure date) RETURNS integer
AS
$$BEGIN
RETURN (
    SELECT count(*)
    FROM booking_leg bl
    JOIN booking b USING (booking_id)
    JOIN passenger p USING (booking_id)
    JOIN flight f USING (flight_id)
    WHERE departure_airport=p_airport_code
    AND scheduled_departure BETWEEN p_departure AND p_departure +1)
;
END;
$$ LANGUAGE plpgsql;

```

Function Execution

To execute a function, we use the SELECT operator. Listing 12-8 demonstrates two possible ways to execute the function num_passengers with the p_flight_id parameter set to 13.

Listing 12-8. Function execution

```
SELECT num_passengers(13);
SELECT * FROM num_passengers(13);
```

For functions that return scalar values, either syntax will produce identical results. Complex types are covered later in this chapter.

It is also worth noting that user-defined scalar functions can be used in SELECT statements, just like internal functions. Recall the function `text_to_numeric` in Listing 12-3. You might wonder why somebody would need to create a user-defined conversion function when PostgreSQL already has three different ways to convert a string to an integer. For the record, these three ways are

- `CAST (text_value AS numeric)`
- `text_value::numeric` (this is alternative syntax for CAST)
- `to_number(text_value, '9999999999')`—using an internal function

Why is a custom conversion function needed? For any of the methods listed in the preceding list, if the input text string contains symbols other than numerals, the attempted conversion results in an error.

To make sure that the conversion function does not fail, we include the *exception processing section* in the function body. The section starts with the `EXCEPTION` keyword; the `WHEN` keyword may identify specific exception types. In this chapter, we will use it only in the form `WHEN OTHERS`, which means all exception types not included in previous `WHEN` conditions. If, as in Listing 12-3, `WHEN OTHERS` is used by itself, it means all exceptions should be processed in the same way.

In Listing 12-3, this means that any conversion error (or, actually, any error) should not fail the function, but instead return `NULL`. Why is it so critical for a function not to fail when a “bad” parameter is passed? Because this function is being used in a SELECT list.

In Chapter 7, we created the materialized view `passenger_passport` (see Listing 7-11). Different columns of this materialized view should contain different data types, but since in the source data, all these fields are text fields, there is not much we can do. Now, if you want to select the `passport_num` as a numeric type, your SELECT might look like this:

```
SELECT
    passenger_id,
    passport_num::numeric AS passport_number
FROM passenger_passport
```

If, in even one instance, the `passport_num` column contains a non-numeric value (e.g., a blank or an empty string), then the whole `SELECT` statement will fail. Instead, we can use the custom function `text_to_integer`:

```
SELECT
    passenger_id,
    text_to_numeric(passport_num) AS passport_number
FROM passenger_passport
```

Let's create one more user-defined function, `text_to_date`, which will transform a string that contains a date to type `date`—see Listing 12-9.

Listing 12-9. Function converting text to date

```
CREATE OR REPLACE FUNCTION text_to_date(input_text text)
    RETURNS date AS
$BODY$
BEGIN
    RETURN input_text::date;
EXCEPTION WHEN OTHERS THEN
    RETURN null::date;
END;
$BODY$
LANGUAGE plpgsql;
```

Now, we can use both functions in Listing 12-10.

Listing 12-10. Using functions in a `SELECT` list

```
SELECT
    passenger_id,
    text_to_integer(passport_num) AS passport_num,
    text_to_date(passport_exp_date) AS passport_exp_date
FROM passenger_passport
```

Although this example seems like a perfect use case for functions in PostgreSQL, in reality it represents a far from ideal solution when it comes to performance—as discussed in the following.

Function Execution Internals

This section explains some specifics of function execution that are unique to PostgreSQL. If you have previous experience with a DBMS like Oracle or MS SQL Server, you might assume a thing or two about function execution that are not true in PostgreSQL.

The first surprise might come when you execute a `CREATE FUNCTION` statement and receive a completion message that looks something like this:

```
CREATE FUNCTION
Query returned successfully in 127 msec.
```

Reading this, you might assume your function does not contain any errors. To illustrate what may go wrong afterward, let's compile the code in Listing 12-11. If you copy and execute this statement, you will receive a successful creation message.

Listing 12-11. Create a function that will complete with no errors

```
CREATE OR REPLACE FUNCTION num_passengers(
    p_airport_code text,
    p_departure date) RETURNS integer
AS $$
BEGIN
RETURN (
    SELECT
        count(*)
    FROM booking_leg bl
    JOIN booking b USING (booking_id)
    JOIN passenger p USING (booking_id)
    JOIN flight f USING (flight_id)
    WHERE airport_code=p_airport_code
        AND scheduled_departure BETWEEN p_date AND p_date +1);
END;
$$ LANGUAGE plpgsql;
```

However, when you try to execute this function

```
SELECT num_passengers('ORD', '2020-07-05')
```

...you will receive an error message:

```
ERROR: column "airport_code" does not exist
```

What went wrong? The function uses `airport_code` instead of `departure_airport`. This is an easy mistake to make, but you might not expect that PostgreSQL would never inform you that you made this mistake in the first place, when you created the function.

Now, if you correct this mistake and run a new `CREATE FUNCTION` statement (see Listing 12-12), you will get yet another error:

```
ERROR: column "p_date" does not exist
```

Listing 12-12. Create a function: one error corrected, one more still there

```
CREATE OR REPLACE FUNCTION num_passengers(
    p_airport_code text,
    p_departure date) RETURNS integer
AS $$
BEGIN
RETURN (
    SELECT
        count(*)
    FROM booking_leg bl
    JOIN booking b USING (booking_id)
    JOIN passenger p USING (booking_id)
    JOIN flight f USING (flight_id)
    WHERE departure_airport =p_airport_code
        AND scheduled_departure BETWEEN p_date AND p_date +1);
END;
$$ LANGUAGE plpgsql;
```

And PostgreSQL is right, since the name of the parameter is `p_departure_date`, not `p_date`. Still, why wasn't this error reported earlier?

During function creation, PostgreSQL performs only an initial parsing pass, during which only trivial syntax errors are detected. Anything deeper will not be detected until execution. If you are fresh from Oracle and assume that when you create a function, it is compiled by the database engine and stored compiled, this is bad news. Not only are functions stored in the form of source code but moreover, in contrast to other DBMSs, functions are interpreted, not compiled.

The PL/pgSQL interpreter parses the function's source text and produces an (internal) instruction tree the first time the function is called within each session. Even then, individual SQL expressions and commands used in the function are not translated

immediately. Only when the execution path reaches a specific command is it analyzed and a *prepared statement* is created. It will be reused if the same function is executed again in the same session. One of the implications of this is that if your function contains some conditional code (i.e., IF THEN ELSE or CASE statements), you may not even discover the syntax errors in your code, if this portion was not reached during execution. We've seen these kinds of unpleasant discoveries made long after the function went into production. To summarize, when you create a PL/pgSQL function

1. No execution plan is saved.
2. No checks for existence of tables, columns, or other functions are performed.
3. You do not know whether your function works or not, until you execute it (often more than one time, if there are multiple code paths).

Another important thing to know about PostgreSQL functions, which can be concluded from the preceding explanation, is that functions are “atomic” in several different ways. First (to the dismay of Oracle users), you can't initiate transactions inside PostgreSQL functions, so in the case of DML statements, it's always “all or nothing.” Second, the PostgreSQL optimizer knows nothing about function execution when it optimizes an execution plan, which includes invocations of user-defined functions. For example, if you execute

```
EXPLAIN SELECT num_passengers(13)
```

...the execution plan will look something like this:

```
"Result (cost=0.00..0.26 rows=1 width=4)"
```

If you need to find out what execution plans are used to execute the SELECT statements inside the function, you will need to supply some actual values in place of parameters and run the EXPLAIN command for each of them.

One of the keywords in the CREATE FUNCTION operator (remember, we didn't list all of them!) is COST. It allows a developer to explicitly set the cost of function execution to be used by the optimizer. The default value is 100, and we do not recommend changing it, unless you have a really compelling reason to do so.

Functions and Performance

With that brief introduction out of the way, it's time to address this book's central concern. How do functions affect performance? Chapter 7 addressed the topic of code factoring and outlined the different implications of code factoring in imperative languages and in SQL. Several possible techniques were covered, and functions were mentioned as deserving a more detailed discussion, which follows.

Why create functions in PostgreSQL? In imperative languages, using functions is the obvious choice: functions increase code readability, facilitate code reuse, and have no negative impact on performance. By contrast, functions in PostgreSQL may increase code readability, but may also decrease code readability and may significantly worsen performance. Note the word “may”; the rest of the chapter concerns ways to use user-defined functions wisely, so that they provide performance benefits rather than performance disaster.

How Using Functions Can Worsen Performance

In a previous section, we created the function `num_passengers(int)`, which calculates the number of passengers on a flight specified by a function parameter. This function works great for a single flight, returning a result within 150 ms.

Let's take a look at what happens if this function is included in a `SELECT` list. Listing 12-13 selects all flights that departed from ORD between June 13 and July 13 and, for each of these flights, calculates a total number of passengers.

Listing 12-13. Using a function in the `SELECT` list decreases performance

```
SELECT
    flight_id,
    num_passengers(flight_id) AS num_pass
FROM flight f
WHERE departure_airport='ORD'
    AND scheduled_departure BETWEEN '2023-07-05' AND '2023-07-13'
```

The execution time for this statement is 13 sec. Now, if instead of using a function, a SQL statement performing the exact same calculations is used (Listing 12-14), the execution time will be only 165 ms.

Listing 12-14. The same results without using a function

```
SELECT
    f.flight_id,
    count(*) AS num_pass
FROM booking_leg bl
JOIN booking b USING (booking_id)
JOIN n passenger p USING (booking_id)
JOIN flight f USING (flight_id)
WHERE departure_airport='ORD'
    AND scheduled_departure BETWEEN '2023-07-05' AND '2023-07-13'
GROUP BY 1
```

Why such a big difference? In Chapter 7, we explained how views and CTEs can work as an optimization fence. The effect is even more pronounced with functions. Since a function is a true black box for the surrounding SQL statement, the only option for PostgreSQL is to execute each function as many times as many rows are selected.

To be precise, some time is saved because for the subsequent function calls from the same session, PostgreSQL uses a prepared statement, but that fact can both speed up and slow down the execution, because the execution plan won't take into account the differences in statistics between function calls.

Even when the difference is not that drastic, and one might argue that a moderate slowdown can be tolerated for ease of code maintenance, we are crossing a threshold of how long a user is willing to wait. And in this case, the SQL inside the function is pretty light and takes milliseconds to be executed.

Fine, we understand that it is not the greatest idea to execute SELECT statements embedded into the SELECT list of another statement. But what about functions, which perform simple data transformations? Like the ones we created for type conversion? In this case, the difference may not be so dramatic until the output size becomes really big, but it is still visible.

Let's compare the execution time for the statement from Listing 12-10 with the execution time for the statement in Listing 12-15.

Listing 12-15. Selecting passport information without type conversion

```
SELECT
    passenger_id,
    passport_num,
    passport_exp_date
FROM passenger_passport
```

Both of them select data from one single table and do not apply any filters, so the only time overhead will be the one incurred from executing the functions in the SELECT list. The `passenger_passport` materialized view contains over 16 million rows. The execution time for the statement in Listing 12-15 is seven seconds. If we apply type casting without calling the function (Listing 12-16)

Listing 12-16. Selecting passport information with type casting

```
SELECT
    passenger_id,
    passport_num::numeric,
    passport_exp_date::date
FROM passenger_passport
```

...the execution time will be nine seconds. Running the statement in Listing 12-10, the execution time will be 40 seconds! Note that in this case the significant execution time increase is a result of exception processing, which creates an implicit checkpoint.

In this particular case, not much can be done to improve performance, except to come up with a better design in the first place, but later in this book, we will review other examples, where some performance improvements are possible.

Any Chance Functions Can Improve Performance?

Having reviewed so many examples where functions affected performance negatively, one might wonder whether there are ever conditions under which functions can improve performance. As in many other cases, it depends.

If we are talking about improving an individual SQL statement's performance, wrapping it in a function can't make it run faster. However, functions can be extremely helpful when what is being optimized is a process.

Functions and User-Defined Types

In all of the function examples so far, the functions we built returned scalar values. Now, let's see what some additional benefits are provided by functions that return user-defined data types.

User-Defined Data Types

In addition to its own rich collection of data types, PostgreSQL allows the creation of a virtually unlimited number of user-defined data types.

User-defined types can be simple or composite. Simple user-defined types include the following categories: domain, enum, and range.

The following are examples of simple type creation:

```
CREATE DOMAIN timeperiod AS tstzrange;
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
CREATE TYPE mood_range AS RANGE...
CREATE TYPE <base type>
```

Just as we can define *arrays* of base types, we can define arrays of user-defined types:

```
DECLARE
v_moods_set mood[];
```

Even more options are available when we create a *composite type*.

A composite type represents a row, or a record. A type definition consists of the sequence of field names and their respective data types. For example, Listing 12-17 defines the type `boarding_pass_record`.

Listing 12-17. Type `boarding_pass_record`

```
CREATE TYPE boarding_pass_record AS (
    boarding_pass_id int,
    booking_leg_id bigint,
    flight_no text,
    departure_airport text,
    arrival_airport text,
    last_name text,
    first_name text,
```

```

        seat text,
        boarding_time timestamptz
    );

```

Now that the type `boarding_pass_record` is defined, we can declare variables of this type, the same as we can declare variables of base types:

```

DECLARE
v_new_boarding_pass_record boarding_pass_record;

```

And moreover, we can create functions, which return *sets of composite types*.

Functions Returning Composite Types

Why is the fact that functions can return sets of composite types so crucial? Why would we want to do it? Recall from Chapter 9 that we need to be able to retrieve the whole object from the database, not just one component after another. Now, everything previously discussed can be put together.

Let's build an example. In Listing 12-18, we present a function that returns all boarding passes for a specified flight.

Listing 12-18. Function returning all boarding passes for the flight

```

CREATE OR REPLACE FUNCTION boarding_passes_flight (p_flight_id int)
RETURNS SETOF boarding_pass_record
AS
$body$
BEGIN
    RETURN QUERY
        SELECT
            pass_id,
            bp.booking_leg_id,
            flight_no,
            departure_airport::text ,
            arrival_airport ::text,
            last_name ,
            first_name ,
            seat,

```

```

        boarding_time
FROM flight f
JOIN booking_leg bl USING (flight_id)
JOIN boarding_pass bp USING(booking_leg_id)
JOIN passenger USING (passenger_id)
WHERE bl.flight_id=p_flight_id;

END;
$body$
LANGUAGE plpgsql;
```

To execute this function, run the following:

```
SELECT * FROM boarding_passes_flight(13);
```

The result of this SELECT is presented in Figure 12-1.

	boarding_pass_id integer	booking_leg_id bigint	flight_no text	departure_airport text	arrival_airport text	last_name text	first_name text	seat text	boarding_time timestamp with time zone
1	215158	15576	1699	KTM	CTU	BLAND	IZAIAH	0B	2023-05-29 23:40:00-05
2	215159	15578	1699	KTM	CTU	JOSEPH	AMELIA	0C	2023-05-29 23:40:00-05
3	215160	15580	1699	KTM	CTU	PETERS	ARIAH	0D	2023-05-29 23:40:00-05
4	215161	15582	1699	KTM	CTU	MC DONALD	EVERLY	0E	2023-05-29 23:40:00-05
5	215162	15584	1699	KTM	CTU	Pi	Liam	0F	2023-05-29 23:40:00-05

Figure 12-1. Result of the execution of the function `boarding_passes_flight`

Now, let’s create another function, which will select just one boarding pass by the `pass_id`. Note that since both functions accept a single integer parameter, overloading won’t be possible in this case. The new function is shown in Listing 12-19.

Listing 12-19. Function that returns one boarding pass

```
CREATE OR REPLACE FUNCTION boarding_passes_pass (p_pass_id int)
RETURNS SETOF boarding_pass_record
AS
$body$
BEGIN
    RETURN QUERY
        SELECT
            pass_id,
            bp.booking_leg_id,
            flight_no,
```

```

        departure_airport::text ,
        arrival_airport ::text,
        last_name ,
        first_name ,
        seat,
        boarding_time
FROM flight f
JOIN booking_leg bl USING (flight_id)
JOIN boarding_pass bp USING(booking_leg_id)
JOIN passenger USING (passenger_id)
WHERE pass_id=p_pass_id;

END;
$body$
LANGUAGE plpgsql;

```

When we execute this function

```
SELECT * FROM boarding_passes_pass(215158);
```

...the result will be a set that consists of only one row, but its structure will be the same (see Figure 12-2).

	boarding_pass_id integer	booking_leg_id bigint	flight_no text	departure_airport text	arrival_airport text	last_name text	first_name text	seat text	boarding_time timestamp with time zone
1	215158	15576	1699	KTM	CTU	BLAND	IZAIAH	0B	2023-05-29 23:40:00-05

Figure 12-2. Result of the execution of the function `boarding_passes_pass`

Why would using these functions improve performance? As we discussed in Chapter 10, applications rarely execute SQL statements directly; instead, they often use SQL statements generated behind the scenes by ORMs. In this case, there is a high likelihood that boarding passes, passengers, and flights are accessible using different methods. That means that most likely, to select the same data returned by the function `boarding_passes_flight`, we will need one method to select departure airport, arrival airport, and scheduled departure for a flight, which is passed as a parameter to this function, another method to select all booking legs for that flight, another method for boarding passes, and yet another for passenger information. If the application developers can be convinced, consolidating this into a single function will be a huge performance improvement.

Selecting all boarding passes for a flight that has 600 passengers with the boarding pass function takes 220 ms, running `SELECT * FROM boarding_passes_flight(13650)`. On the other hand, any individual `SELECT` from any table takes around 150 ms. Since each process returns data to the application, making a roundtrip, the execution time is summed up—using multiple calls will very quickly exceed the execution time of the function.

Previously, we learned that for scalar functions there is no difference between the syntaxes `SELECT * FROM function_name` and `SELECT function name`. But when a function returns a composite type, there is a difference.

Figure 12-1 shows the results when running
`SELECT * FROM boarding_passes_flight(13)`

Figure 12-3 shows the results of
`SELECT boarding_passes_flight(13)`

	boarding_passes_flight boarding_pass_record	
1	(215158,15576,1699,KTM,CTU,BLAND,IZAIAH,0B,"2023-05-29 23:40:00-05")	
2	(215159,15578,1699,KTM,CTU,JOSEPH,AMELIA,0C,"2023-05-29 23:40:00-05")	
3	(215160,15580,1699,KTM,CTU,PETERS,ARIAH,0D,"2023-05-29 23:40:00-05")	
4	(215161,15582,1699,KTM,CTU,"MC DONALD",EVERLY,0E,"2023-05-29 23:40:00-0...	
5	(215162,15584,1699,KTM,CTU,Pi,Liam,0F,"2023-05-29 23:40:00-05")	

Figure 12-3. *Function results as a set of records*

Using Composite Types with Nested Structure

Can we use composite types as elements of other composite types? Yes, PostgreSQL allows it.

In Figure 11-4, we presented the structure of a complex object `booking_record`. One of its components is a complex object `booking_leg_record`. To build the representation of this object as a composite type, begin by creating a `flight_record` type and a `boarding_pass_record` type, and then proceed with creating a `booking_leg_record` type, as shown in Listing 12-20.

Listing 12-20. More record type definitions

```

CREATE TYPE flight_record AS(
    flight_id int,
    flight_no text,
    departure_airport_code text,
    departure_airport_name text,
    arrival_airport_code text,
    arrival_airport_name text,
    scheduled_departure timestamptz,
    scheduled_arrival timestamptz
);
CREATE TYPE booking_leg_record AS(
    booking_leg_id int,
    leg_num int,
    booking_id int,
    flight flight_record,
    boarding_passes boarding_pass_record[]
);

```

The `booking_leg_record` type contains as one of its elements a composite type `flight_record` and as another component the array of `boarding_pass_record` elements.

Looks like we solved the problem stated in Chapter 11: we can create composite types with nested structure and create functions that return such objects. However, there are still plenty of problems to solve.

To illustrate the remaining problems, let's create a function that will return the whole object `booking_leg_record` using `booking_leg_id`. The code for this function is presented in Listing 12-21.

Listing 12-21. Function returning a complex object with nested structure

```

CREATE OR REPLACE FUNCTION booking_leg_select (p_booking_leg_id int)
RETURNS SETOF booking_leg_record
AS
$body$
BEGIN
    RETURN QUERY
        SELECT

```

```

        bl.booking_leg_id,
        leg_num,
        bl.booking_id,
    (SELECT row(
        flight_id,
        flight_no,
        departure_airport,
        da.airport_name,
        arrival_airport,
        aa.airport_name ,
        scheduled_departure,
        scheduled_arrival)::flight_record
    FROM flight f
    JOIN airport da on da.airport_code=departure_airport
        JOIN airport aa on aa.airport_code=arrival_airport
        WHERE flight_id=bl.flight_id
    ),
    (SELECT array_agg (row(
        pass_id,
        bp.booking_leg_id,
        flight_no,
        departure_airport ,
        arrival_airport,
        last_name ,
        first_name ,
        seat,
        boarding_time)::boarding_pass_record)
    FROM flight f1
    JOIN boarding_pass bp ON f1.flight_id=bl.flight_id
        AND bp.booking_leg_id=bl.booking_leg_id
    JOIN passenger p ON p.passenger_id=bp.passenger_id)
    FROM booking_leg bl
    WHERE bl.booking_leg_id=p_booking_leg_id

```

```

;
END;
$body$ language plpgsql;

```


Don't be put off by the preceding massive function—it is long, but not too complex. Let's take a closer look.

The main SELECT extracts data from the `booking_leg` table, using the value of the function parameter as a search criterion. The first three elements of the record—`booking_leg_id`, `leg_num`, `booking_id`—come directly from the table `booking_leg`. The next element of the record is `flight_record`, where `flight_id` is the `flight_id` from the selected booking leg. This condition is set in the WHERE clause of the inner SELECT:

```
WHERE flight_id=bl.flight_id
```

We select the information about the flight, which is referenced in the selected booking leg.

The internal function `row()` builds the row from the set of elements, and this row is cast to the type `flight_record`, which is the type expected in the `booking_leg_record`.

The last element of the `booking_leg_record` is an array of boarding passes—as many passes as there are passengers in this reservation. Let's take a closer look at this inner SELECT:

```
(SELECT array_agg (row(
                                pass_id,
                                bp.booking_leg_id,
                                flight_no,
                                departure_airport ,
                                arrival_airport,
                                last_name ,
                                first_name ,
                                seat,
                                boarding_time)::boarding_pass_record)
FROM flight f1
JOIN  boarding_pass bp ON f1.flight_id=bl.flight_id
                        AND bp.booking_leg_id=bl.booking_leg_id
JOIN passenger p ON p.passenger_id=bp.passenger_id)
```

The first thing you notice about this select is that it is essentially the same select as we used in the `boarding_pass_flight` function. The differences are the following:

- There is no need to join with the `booking_leg` table, since it was already selected in the outer `SELECT`. We still need information from the `flight` table, but we can use the `flight_id` from the selected booking leg. This way, there is a Cartesian product with one line from the table `flight`.
- Similarly, for the boarding pass, there is no join with the `booking_leg` table; we just use the `booking_leg_id`, which is already there.

Finally, we use the internal function `array_agg()` to create a single *array of records* that is expected as the last element of the `booking_leg_record`.

Note The preceding is only one of multiple ways to build an object with nested structure. In subsequent chapters, we will present alternative ways, which might be more useful in other circumstances.

And now, here is the bad news. We put in so much effort to create this function, and now, when we execute it, the results are somewhat disappointing. Execute

```
SELECT * FROM booking_leg_select (17564910)
```

The result is shown in Figure 12-4.



The screenshot shows a single row of data from a PostgreSQL query. The row contains four columns: `booking_leg_id` (integer), `leg_num` (integer), `booking_id` (integer), and `flight` (flight_record). The values are 17564910, 2, 232346, and a complex nested structure representing a flight record. The nested structure includes details like flight number (13650), origin (JFK), destination (CDG), and departure/arrival times.

booking_leg_id	leg_num	booking_id	flight
17564910	2	232346	{13650,1245,JFK,"John F Kennedy International Airport",CDG,"Charles de Gaulle International Airport","2023-06-02 00:20:00-05","2023-06-02 06:15:00..."}

Figure 12-4. Returned complex object with nested structure

The result set looks exactly as we wanted it to look, but notice one important detail. For the scalar elements, PostgreSQL retains the element names (same as if we would select from a table), but when it comes to the elements that are complex objects themselves, their structure is not revealed. Note that, internally, PostgreSQL still retains the notion of what is the structure of the inner type, but it does not communicate it to the upper level.

Why is this a problem? Chapter 11 covered ORM pitfalls and sketched out a hypothetical solution in Figure 11-3. At that time, we did not discuss any specifics of how this goal could be achieved, but with functions that can return complex types, a solution seems to be reachable. However, when neither an element name nor its type can be identified by the application, the function output becomes useless, at least for the purpose of being called from an application directly.

The solution is in Chapter 14, but for now, let's focus on functions that return records without nested structure.

Functions and Type Dependencies

Chapter 7 mentioned dependencies in the context of views and materialized views. For materialized views, the definition cannot be altered without dropping the object first. This, in turn, means all dependent objects must be dropped and recreated, even if the names and the number of columns in the materialized view didn't change. If these dependent objects are, in turn, used in other views or materialized views, their dependent objects have to be dropped as well.

This may result in some highly undesirable consequences. We've observed situations in production systems where one change resulted in a cascade drop of over 60 dependent objects, which had to be rebuilt following a particular order.

Fortunately, we do not have this problem with functions. Since SQL statements in the function body are not parsed during function creation, there are no dependencies on tables, views or materialized views, or other functions and stored procedures, which are used in the function body. For this reason, functions need only be recreated when needed as the result of an actual change, not simply due to a cascade drop.

However, functions create a new type of dependencies: functions depend on their returned types, including user-defined types. Just as with materialized views, user-defined data types cannot be modified without being dropped first. To drop a type, all other user-defined types that include it as an element and all functions that depend on that type must be dropped. This might sound like an even worse problem, but actually, it is exactly the right problem to have. If a user-defined type is modified, some of its elements must have been added, removed, or changed. That, in turn, means that SELECT statements that return that type of record must be revised, so the functions should be dropped.

In addition, unlike the creation of a materialized view, which may take some time, creating a function is nearly instantaneous.

Data Manipulation with Functions

So far, this chapter has only considered functions that select data. But PL/pgSQL functions allow any SQL command, including DML functions.

Listing 12-22 is a function that issues a new boarding pass to a passenger.

Listing 12-22. Create a new boarding pass

```

CREATE OR REPLACE FUNCTION issue_boarding_pass(
    p_booking_leg_id int,
    p_passenger_id int,
    p_seat text,
    p_boarding_time timestampz)
RETURNS SETOF boarding_pass_record
AS
$body$
DECLARE
    v_pass_id int;
BEGIN
    INSERT INTO boarding_pass
        (passenger_id,
         booking_leg_id,
         seat,
         boarding_time,
         update_ts)
    VALUES (
        p_passenger_id,
        p_booking_leg_id,
        p_seat,
        p_boarding_time,
        now()) RETURNING pass_id INTO v_pass_id;
    RETURN QUERY
        SELECT * FROM boarding_passes_pass(v_pass_id);
END;
$body$
LANGUAGE plpgsql;

```

Note the call to the function `boarding_passes_pass` in this function body. This function was created earlier, but even if it didn't exist, the `CREATE FUNCTION` operator wouldn't signal an error until this function is executed. There are pros and cons to this behavior. It gives more flexibility during development, but it can also create issues because the fact that the embedded function was removed or is not working properly might go unnoticed. Executing this function is the same as other functions:

```
SELECT * FROM issue_boarding_pass(175820,462972, '22C', '2023-06-16
21:45'::timestampz)
```

Note that this execution does not make much sense because the flight departed in the past, so it is present here for illustrative purposes only. Figure 12-5 presents the result of this execution—the data has the same format as for other functions that return boarding passes.

	boarding_pass_id integer	booking_leg_id bigint	flight_no text	departure_airport text	arrival_airport text	last_name text	first_name text	seat text	boarding_time timestamp with time zone
1	25293500	175820	68	AMS	FRA	WOOD	KAITLYN	22C	2023-06-16 21:45:00-05

Figure 12-5. DML function returning a user-defined type

When creating this function, we made some assumptions that we would not hold up in real life. For example, the function doesn't check whether a boarding pass for that passenger and that flight was already issued, doesn't check seat availability against the seat chart, and does not capture possible errors on INSERT. In a production environment, this function would be much more complex.

Functions and Security

In this book, we do not cover data access control/permissions in PostgreSQL, mostly because this topic is not related to performance. However, we will cover a little bit about setting up security for PostgreSQL functions and a surprising link between function security settings and performance.

One of the parameters in the CREATE FUNCTION operator that was not covered earlier is SECURITY. This parameter has only two allowed values: INVOKER and DEFINER. The former is a default value; it indicates that the function will be executed using the set of privileges of the user who calls the function. That means that in order to be able to execute a function, a user should have relevant access to all the database objects that are used in the function body. If we explicitly specify SECURITY DEFINER, a function will be executed with the permissions of the user who created the function. Note that in contrast to other database object permissions, the execution privilege for any function is granted by default to PUBLIC.

Many of you (as well as us, your authors) have been in a situation where a power business user needs to have access to some critical data, but you do not want to give them READ ALL access, because you are not entirely sure about their SQL skills and whether their queries might bring the whole system down.

In this case, a compromise might be in order—you would create a function that pulls all the necessary data using a performant query, create this function with the SECURITY DEFINER parameter, remove execution permission from everybody (from PUBLIC), and then give this power user the execution permission. The sequence of actions is presented in Listing 12-23.

Listing 12-23. Usage of the SECURITY DEFINER function

```
CREATE FUNCTION critical_function (par1 ...)
RETURNING SETOF...
AS $FUNC$
...
END:
$FUNC$
LANGUAGE plpgsql
SECURITY DEFINER;
--
REVOKE EXECUTE ON critical_function (par1 ...)
FROM public;
GRANT EXECUTE ON critical_function (par1 ...)
TO powerbusinessuser;
```

What About Business Logic?

In cases when you can convince your application developers to use functions to communicate with the database, the performance gains are dramatic. The sheer fact of eliminating multiple roundtrips can easily improve application performance tens or even hundreds of times, when we measure the application response time rather than the database response time.

One of the most serious blockers on this road to success is the concept of *business logic*. One of the definitions (from Investopedia.com) reads like this:

Business logic is the custom rules or algorithms that handle the exchange of information between a database and user interface. Business logic is essentially the part of a computer program that contains the information (in the form of business rules) that defines or constrains how a business operates.

Business logic is often considered a separate application layer, and when we put “too much logic” into database functions, it makes application developers unhappy. We spent a considerable amount of time trying to find common ground with business and application developers alike. The result of these discussions can be summarized as follows:

- We need some business logic to execute joins and selects.
- Selected result transformations and manipulations do not have to be executed on the database side.

In practice, this means that when deciding what can go to the database and what has to stay in the application, a decisive factor is whether bringing the dependencies into the database would improve performance (facilitate joins or enable the use of indexes). If so, the logic is moved into a function and considered “database logic”; otherwise, data is returned to the application for further processing of business logic.

For example, for the airline reservation application, a function can be created to return available trips, that is, potential bookings. The parameters of this function include the departure city, the destination, the trip start date, and the return date. To be able to retrieve all possible trips efficiently, the function needs to know how the tables `airport` and `flight` can be joined and how to calculate the duration of the flight. All this information belongs to database logic.

However, we do not want the function to make a final decision regarding which trip to select. Final selection criteria may vary and are processed by the application; they belong to business logic.

Applying this criterion consistently can be quickly incorporated into the regular development cycle and encourages developing applications “right right away.”

Functions in OLAP Systems

By this time, we hope that we convinced you that using PostgreSQL functions in OLTP systems is beneficial. What about OLAP?

Unless you’ve tried it, you might not know that many reporting tools, including Cognos, Business Objects, and Looker, can present the results of a function. In fact, executing a function that returns a set of records is similar to performing `SELECT * FROM <some table>`.

However, the fact that software can do something doesn’t mean that it should. So when should you use functions in an OLAP environment?

Parameterizing

A view or materialized view can't be parameterized. This might not pose any problem if we want to run a report for the most recent date, for yesterday, for last week, and so on, because we can utilize such internal functions as `CURRENT_DATE` or `CURRENT_TIMESTAMP`, but if we need to rerun any report for any of the past time intervals, it won't be an easy task without making some changes to the view. For example, we might have a view

```
CREATE VIEW recent_flights_v AS
SELECT flight_id,
       departure_airport,
       arrival_airport,
       scheduled_departure
FROM flight
WHERE scheduled_departure BETWEEN CURRENT_DATE-7
AND CURRENT_DATE
```

If you need to run the same query for a different date interval, you will need to create another view (or recreate the existing view). But if this `SELECT` is packaged into the function

```
CREATE OR REPLACE FUNCTION recent_flights(p_date date)
RETURNS SETOF recent_flights_record AS
$BODY$
BEGIN
    RETURN QUERY
        SELECT flight_id,
               departure_airport,
               arrival_airport,
               scheduled_departure
        FROM flight
        WHERE scheduled_departure BETWEEN p_date-7
        AND p_date;
END; $BODY$;
```

...you can simply execute it with different parameters:

```
SELECT * FROM recent_flights(CURRENT_DAY)
SELECT * FROM recent_flights('2023-08-01')
```


No Explicit Dependency on Tables and Views

If a report is executed as a call to a function, it can be optimized without the necessity to drop and recreate it. Moreover, the underlying tables can be modified, or we can end up using completely different tables, all invisible to the end user.

Ability to Execute Dynamic SQL

Functions allow building SQL statements based on the values of parameters. The SQL language does not have an option of building the statement on the fly and then executing it. You can build the SQL statement in the application and then send it to the database to execute, or you can build it inside the `plpgsql` function. This is another exceptionally powerful feature of PostgreSQL, which is often underused and which is discussed in more detail in Chapter 13.

Stored Procedures

In contrast to other DBMSs, PostgreSQL didn't have stored procedures for some time, much to the disappointment of early adopters coming from commercial systems. As for us, your authors, we were especially frustrated with the atomic nature of functions, which does not allow any transaction management, including committing of intermediate results.

Functions with No Results

For a while, PostgreSQL developers had no option rather than to use functions in place of store procedures. You could do it using functions that return `VOID`, like

```
CREATE OR REPLACE function cancel_flight (p_flight_id int) RETURNS VOID
AS <...>
```

Also, there is an alternative way to execute functions when calling them from another function or procedure:

```
PERFORM issue_boarding_pass(175820,462972, '22C', '2020-06-16
21:45'::timestampz)
```

The preceding way will execute the function and create a boarding pass, but it won't return the result.

Functions and Stored Procedures

The difference between functions and stored procedures is that procedures do not return any values; thereby, we do not specify a return type. Listing 12-24 presents the CREATE PROCEDURE command, which is very similar to the CREATE FUNCTION command.

Listing 12-24. CREATE PROCEDURE command

```
CREATE PROCEDURE procedure_name (par_name1 par_type1, ...)
AS
<procedure body>
LANGUAGE procedure language;
```

The syntax of the procedure body is the same as that of the function, except there is not a need for a RETURN type. Also, all the preceding sections on function internals apply to stored procedures, as well. To execute a stored procedure, the CALL command is used:

```
CALL cancel_flight(13);
```

Transaction Management

The most important difference between how functions and stored procedures are executed is that you can commit or roll back a transaction within a procedure body.

At the start of the procedure execution, a new transaction starts, and any COMMIT or ROLLBACK command within a function body will terminate the current transaction and start a new one. One of the use cases is the bulk data load. We find it beneficial to commit changes in reasonably sized portions, for example, every 50,000 records. The structure of the stored procedure might look like Listing 12-25.

Listing 12-25. Example of a stored procedure with transactions

```
CREATE PROCEDURE load_with_transform()
AS $load$
DECLARE
    v_cnt int:=0;
    v_record record;
BEGIN
    FOR v_record IN (SELECT * FROM data_source) LOOP
```

```

        PERFORM transform (v_rec.id);
        CALL insert_data (v_rec.*);
        v_cnt:=v_cnt+1;
        IF v_cnt>=50000 THEN
            COMMIT;
            v_cnt:=0;
        END IF;
    END LOOP;
    COMMIT;
END;
$load$ LANGUAGE plpgsql;

```

In this example, data is processed before loading and COMMIT when we process 50,000 records. An additional commit upon exiting the loop is necessary for the remaining records, processed after the last in-loop commit.

Exception Processing

Same as with functions, you can include instructions on what to do if certain processing exceptions occur. In Listing 12-3, we provided an example of exception processing in a function. Similar exception processing can be performed in procedures.

In addition, it is possible to create inner blocks inside of a function or procedure body and to have a different exception processing in each of them. The procedure body structure for this case is shown in Listing 12-26.

Listing 12-26. Nested blocks in the procedure body

```

CREATE PROCEDURE multiple_blocks AS
$mult$
BEGIN
    ---case #1
    BEGIN
        <...>
        EXCEPTION WHEN OTHERS THEN
            RAISE NOTICE 'CASE#1';
        END; --case #1

```

```

    ---case #2
    BEGIN
    <...>
    EXCEPTION WHEN OTHERS THEN
        RAISE NOTICE 'CASE#2';
    END; --case #2
---case #3
    BEGIN
    <...>
    EXCEPTION WHEN OTHERS THEN
        RAISE NOTICE 'CASE#3';
    END; --case #3
END; ---proc
$mult$ LANGUAGE plpgsql;

```

Note that `BEGIN` in the procedure body is different from the `BEGIN` command that starts a transaction.

Summary

Functions and stored procedures in PostgreSQL are exceptionally powerful tools that are all but ignored by many database developers. They can both drastically improve and drastically worsen performance and can be successfully used in both OLTP and OLAP environments. Chapter [14](#) will provide more guidance on making decisions regarding the function usage.

This chapter serves as a sneak peek of various ways functions can be used. Consult PostgreSQL documentation for more details on how to define and use functions and stored procedures.

CHAPTER 13

Dynamic SQL

What Is Dynamic SQL

Dynamic SQL is any SQL statement that is first built as a text string and then executed using the EXECUTE command. An example of dynamic SQL is shown in Listing 13-1. Dynamic SQL is underused in most RDBMSs, but especially in PostgreSQL. The recommendations in this chapter go against the grain of many database textbooks, but all suggestions are based strictly on our practical experience.

Listing 13-1. Dynamic SQL

```
DECLARE
    v_sql text;
    cnt int;
BEGIN
    v_sql:=$$SELECT count(*)
                FROM booking
                WHERE booking_ref='0Y7W22'$$;
    EXECUTE v_sql into cnt;
    <...>
```

Why Dynamic SQL Works Better in Postgres

What is so special about PostgreSQL relative to other DBMSs, to make the recommendations here stray so far from conventional wisdom? Consider the following.

First, in PostgreSQL, execution plans are not cached even for *prepared queries* (i.e., queries that are prepared, analyzed, and rewritten using the PREPARE command). That means that optimization always happens immediately before execution.

Second, the optimization step in PostgreSQL happens later than in other systems. For example, in Oracle, the execution plan for a parameterized query is always prepared for a generic query, even if the specific values are there. Moreover, a plan with binding variables is cached for future usage if the same query with different values is executed. The optimizer takes table and index statistics into account but does not take into account the specific values of parameters. PostgreSQL does the opposite. The execution plan is generated for specific values.

As mentioned earlier, dynamic queries are unfairly neglected in other DBMSs as well. That's primarily because for long-running queries (dozens of seconds or more), the overhead mostly is negligible.

What About SQL Injection?

Often, the suggestion to use dynamic SQL for better performance is met with alarm: What about SQL injection? Of course, we've all heard horror stories about stolen passwords and deleted data, because a malicious actor passed a dangerous command instead of data into a form. Of course, this is a real risk, and there's more than one way for hackers to get access to data they should not have access to. However, there are guardrails available to using dynamic SQL that minimize risks.

In cases when parameter values for a function call are obtained from the database directly (e.g., referencing IDs), SQL injection is impossible. Values obtained from user input must be sanitized with PostgreSQL functions (`quote_literal`, `quote_ident`, etc. or `format`)—examples of this are demonstrated later in this chapter. User inputs can also benefit from validation on the application side.

How to Use Dynamic SQL for an Optimal Execution Plan

As mentioned in Chapter 12, PostgreSQL may reuse execution plans for parametrized SQL statements issued from functions, which in some cases may cause poor performance. To ensure that the execution plan is built optimally, we can build dynamic SQL inside a function and then execute it rather than pass parameter values as binding variables. Let's look at an example.

Recall the query in Listing 6-6, which has two selection criteria: departure airport country and the last time the booking was updated. Chapter 6 demonstrated how PostgreSQL modifies the execution plan depending on specific values of these parameters.

In this chapter, we see what happens with this query if it is executed inside a function.

Let's start by creating a return type (see Listing 13-2).

Listing 13-2. Create a return type

```
DROP TYPE IF EXISTS booking_leg_part ;
CREATE TYPE booking_leg_part AS(
    departure_airport char (3),
    booking_id int,
    is_returning boolean
);
```

Now, let's create a function with two parameters: ISO country code and the timestamp of the last update. This function is shown in Listing 13-3.

Listing 13-3. SQL from Listing 6-6, packaged in a function

```
CREATE OR REPLACE FUNCTION select_booking_leg_country (
    p_country text,
    p_updated timestamptz)
RETURNS SETOF booking_leg_part
AS
$body$
BEGIN
RETURN QUERY
SELECT
    departure_airport,
    booking_id,
    is_returning
FROM booking_leg bl
JOIN flight f USING (flight_id)
WHERE departure_airport IN
```

```

        (SELECT
            airport_code
        FROM airport
        WHERE iso_country=p_country
        )
    AND bl.booking_id IN
        (SELECT
            booking_id
        FROM booking
        WHERE update_ts>p_updated
        )
;
END; $body$
LANGUAGE plpgsql;

```

Chapter 6 demonstrated how PostgreSQL chooses different execution plans depending on the values for country and timestamp search parameters and how this influences execution time.

Since functions in PostgreSQL (just as in other systems) are atomic, we can't run the EXPLAIN command to see the execution plan for the function (to be precise, the EXPLAIN will be executed, but the only thing it will show will be the execution itself), but since the expected response time for the query is known, we can get a good idea what's going on under the hood.

Recall that previously, executing the statement in Listing 13-4 resulted in an execution time of about 9.5 seconds, with two hash joins executed.

Listing 13-4. SELECT with two hash joins

```

SELECT
    departure_airport,
    booking_id,
    is_returning
FROM booking_leg bl
JOIN flight f USING (flight_id)
WHERE departure_airport IN
    (SELECT
        airport_code

```



```

        FROM airport WHERE iso_country='US'
    )
    AND bl.booking_id IN
    (SELECT
        booking_id
    FROM booking
    WHERE update_ts>'2023-07-01'
    )

```

Recall also that by moving the bound on update_ts closer to the dataset’s “current date” of August 17, initially, the execution time doesn’t change significantly. Execution time with update_ts>'2023-08-01' will still be about 7.9 seconds, with a reduction attributable to a smaller intermediate dataset. The execution plan for that case is shown in Figure 13-1.

	QUERY PLAN	
	text	
1	Hash Join (cost=169581.38..630554.84 rows=286036 width=9)	
2	Hash Cond: (bl.booking_id = booking.booking_id)	
3	-> Hash Join (cost=18963.54..433030.38 rows=3645943 width=9)	
4	Hash Cond: (bl.flight_id = f.flight_id)	
5	-> Seq Scan on booking_leg bl (cost=0.00..310506.66 rows=17893566 width=9)	
6	-> Hash (cost=17223.52..17223.52 rows=139202 width=8)	
7	-> Hash Join (cost=20.41..17223.52 rows=139202 width=8)	
8	Hash Cond: (f.departure_airport = airport.airport_code)	
9	-> Seq Scan on flight f (cost=0.00..15398.78 rows=683178 width=8)	
10	-> Hash (cost=18.65..18.65 rows=141 width=4)	
11	-> Seq Scan on airport (cost=0.00..18.65 rows=141 width=4)	
12	Filter: (iso_country = 'US'::text)	
13	-> Hash (cost=143353.65..143353.65 rows=442735 width=8)	
14	-> Bitmap Heap Scan on booking (cost=8291.63..143353.65 rows=442735 width=8)	
15	Recheck Cond: (update_ts > '2023-08-01 00:00:00-05':timestamp with time zone)	
16	-> Bitmap Index Scan on booking_update_ts (cost=0.00..8180.94 rows=442735 width=...)	
17	Index Cond: (update_ts > '2023-08-01 00:00:00-05':timestamp with time zone)	

Figure 13-1. Execution plan with two hash joins

Eventually, as the value of `update_ts` approaches August 17, PostgreSQL will choose index access, and for the query in Listing 13-5, the execution time is 1.8 seconds.

Listing 13-5. One hash join is replaced with a nested loop

```
SELECT departure_airport, booking_id, is_returning
FROM booking_leg bl
JOIN flight f USING (flight_id)
WHERE departure_airport IN
      (SELECT airport_code
       FROM airport WHERE iso_country='US')
AND bl.booking_id IN
      (SELECT booking_id FROM booking
       WHERE update_ts>'2023-08-14')
```

The execution plan for this case is shown in Figure 13-2.

1	Hash Join (cost=29828.94..401980.77 rows=6682 width=9)
2	Hash Cond: (f.departure_airport = airport.airport_code)
3	-> Nested Loop (cost=29808.52..401873.75 rows=32795 width=9)
4	-> Hash Join (cost=29808.10..387285.40 rows=32795 width=9)
5	Hash Cond: (bl.booking_id = booking.booking_id)
6	-> Seq Scan on booking_leg bl (cost=0.00..310506.66 rows=17893566 width=9)
7	-> Hash (cost=29678.81..29678.81 rows=10343 width=8)
8	-> Bitmap Heap Scan on booking (cost=196.59..29678.81 rows=10343 width=8)
9	Recheck Cond: (update_ts > '2023-08-14 00:00:00-05'::timestamp with time zo...
10	-> Bitmap Index Scan on booking_update_ts (cost=0.00..194.00 rows=10343 ...)
11	Index Cond: (update_ts > '2023-08-14 00:00:00-05'::timestamp with time zo...
12	-> Index Scan using flight_pkey on flight f (cost=0.42..0.44 rows=1 width=8)
13	Index Cond: (flight_id = bl.flight_id)
14	-> Hash (cost=18.65..18.65 rows=141 width=4)
15	-> Seq Scan on airport (cost=0.00..18.65 rows=141 width=4)
16	Filter: (iso_country = 'US'::text)

Figure 13-2. Execution plan with one hash join and one nested loop

Also, if we run the same SELECT with the `iso_country='CZ'`, PostgreSQL will produce yet another differing execution plan, as shown in Figure 13-3. In this case, the execution time will be around 200 ms.

	QUERY PLAN text
1	Nested Loop (cost=13.49..18917.18 rows=8349 width=9)
2	-> Nested Loop (cost=13.06..6615.00 rows=25858 width=9)
3	-> Nested Loop (cost=12.62..3022.73 rows=987 width=8)
4	-> Seq Scan on airport (cost=0.00..18.65 rows=1 width=4)
5	Filter: (iso_country = 'CZ'::text)
6	-> Bitmap Heap Scan on flight f (cost=12.62..2993.50 rows=1058 width=8)
7	Recheck Cond: (departure_airport = airport.airport_code)
8	-> Bitmap Index Scan on flight_departure_airport (cost=0.00..12.36 rows=1058 width=...
9	Index Cond: (departure_airport = airport.airport_code)
10	-> Index Scan using booking_leg_flight_id on booking_leg bl (cost=0.44..2.79 rows=85 width=...
11	Index Cond: (flight_id = f.flight_id)
12	-> Index Scan using booking_pkey on booking (cost=0.43..0.48 rows=1 width=8)
13	Index Cond: (booking_id = bl.booking_id)
14	Filter: (update_ts > '2023-07-01 00:00:00-05'::timestamp with time zone)

Figure 13-3. Execution plan with nested loops only

With these figures as a reference, let's examine how the function version of the query performs.

Let's try to replicate the same behavior observed in Chapter 6 for a long query with different search conditions and execute the statements shown in Listing 13-6.

Listing 13-6. Examples of function calls

```
#1
SELECT * FROM select_booking_leg_country('US', '2023-07-01');
#2
SELECT * FROM select_booking_leg_country('US', '2023-08-01');
```

#3

```
SELECT * FROM select_booking_leg_country('US', '2023-08-14');
```

#4

```
SELECT * FROM select_booking_leg_country('CZ', '2020-08-01');
```

The execution times observed will differ depending on what parameters are passed to the function during the first three to five calls. As a result, the execution time for statement #1, which should take around nine seconds, may vary from nine seconds up to one minute, depending on the sequence of calls. You can even open two or three connections to your local PostgreSQL and try to execute these calls in different orders.

Why is the behavior of the function so inconsistent? Recall Chapter 12, which stated that PostgreSQL *may* save the execution plan of the prepared statement and when a function is called for the first time in a session, each SQL statement that it reaches during the execution will be evaluated and the execution plan will be optimized and then it *may* be cached for subsequent executions.

We are purposely not describing a specific behavior with each sequence of calls, because it is not guaranteed. And while “not guaranteed” may be acceptable for a training database, it is definitely not acceptable in a production environment, especially when an OLTP system implements a policy that caps maximum wait time and aborts transactions when the wait time exceeds that limit. Also, let’s not forget about connection pools. Practically any web application initiates a certain number of connections, called a connection pool, and these connections are used by application users’ processes, most often, for a single SQL statement. As a result, one PostgreSQL session will almost certainly execute each function with very different sets of parameters.

To guarantee that each time a function is executed *the execution plan will be evaluated and optimized for the specific values it’s called with*, we create functions that execute dynamic SQL.

Listing 13-7 shows the function `select_booking_leg_country_dynamic` that executes the same SQL as the `select_booking_leg_country` function. The only difference is that the former function constructs a SELECT statement inside the function and then executes it.

Listing 13-7. A function that executes dynamic SQL

```
CREATE OR REPLACE FUNCTION select_booking_leg_country_dynamic(
    p_country text,
    p_updated timestamptz)
RETURNS setof booking_leg_part
```

```

AS
$body$
BEGIN
RETURN QUERY
EXECUTE $$
SELECT
    departure_airport,
    booking_id,
    is_returning
FROM booking_leg bl
JOIN flight f USING (flight_id)
WHERE departure_airport IN
    (SELECT
        airport_code
        FROM airport
        WHERE iso_country=$$|| quote_literal(p_country) ||
    $$ )
AND bl.booking_id IN
    (SELECT
        booking_id
        FROM booking
        WHERE update_ts>$$|| quote_literal(p_updated)||$$)$$;
END;
$body$ LANGUAGE plpgsql;

```

This function accepts the same set of parameters as `select_booking_leg_country` and returns the same result. But observe that its execution time for each set of parameters is consistent, which is exactly what we want in production systems.

Why did this behavior change? Since the SQL is built immediately prior to execution, the optimizer does not use a cached plan. Instead, it evaluates the execution plan for each execution. It may seem that this would take extra time, but in practice, the opposite happens. Planning time is under 100 ms, and it pays off with a better execution plan, which saves significantly more time.

Also note that this function uses the `quote_literal()` function to guard against SQL injection.

This is the first but not the only reason to use dynamic SQL in functions. More cases in support of this practice follow later in this chapter.

How to Use Dynamic SQL in OLAP Systems

The title of this section may be misleading. The technique demonstrated in the following can be used in any system; however, the most impressive results are observed when the result set is large. The larger the result set is, the more pronounced the benefits.

Let's assume that for statistical analysis, we need to sort passengers by age. A function to define age categories is defined in Listing 13-8.

Listing 13-8. A function that assigns the age category

```
CREATE OR REPLACE FUNCTION age_category (p_age int)
RETURNS TEXT language plpgsql AS
$body$
BEGIN
    RETURN (CASE
        WHEN p_age <= 2 then 'Infant'
        WHEN p_age <=12 then 'Child'
        WHEN p_age < 65 then 'Adult'
        ELSE 'Senior'
    END);
END; $body$;
```

If this function is used for statistical reports, we might need to calculate an age category for every passenger. In Chapter 12, we mentioned that executing functions in the SELECT list may slow things down, but the functions were more complex. The age_category function performs a very simple substitution. Still, function invocation takes time. Thus,

```
SELECT
    passenger_id,
    age_category(age)
FROM passenger
```

takes 6.5 seconds to execute, while

```
SELECT passenger_id,
    CASE
        WHEN age <= 2 then 'Infant'
```

```

        WHEN age <=12 then 'Child'
        WHEN age < 65 then 'Adult'
        ELSE 'Senior'
    END from passenger

```

takes 3.5 seconds.

In this particular case, using a function is not really imperative, because it is used only once, and even one of the biggest tables in Postgres Air, `passenger`, has only 16 million rows. In real analytical queries, the number of rows to process might be hundreds of millions, and multiple category-assigning functions might be needed. In one real-life scenario, execution time with functions was four hours, while execution time with just one function substituted by a direct CASE operator was less than 1.5 hours.

Does this mean we want to avoid using functions in the SELECT list at all costs? There may be a reason our analytics team wants to package the age category assignment in the function. Most likely, they are going to use this function in multiple selects and with different tables and want their queries to be resilient and consistent if age categories ever change.

A more performant solution that retains the maintainability of the function is to create a different function, which contains *part of the code as text*, as in Listing 13-9.

Listing 13-9. A function that builds dynamic SQL

```

CREATE OR REPLACE FUNCTION age_category_dyn (p_age text)
RETURNS text language plpgsql AS
$body$
BEGIN
    RETURN ($$CASE
        WHEN $$||p_age ||$$ <= 2 THEN 'Infant'
        WHEN $$||p_age ||$$<= 12 THEN 'Child'
        WHEN $$||p_age ||$$< 65 THEN 'Adult'
        ELSE 'Senior'
    END$$);
END; $body$;

```

Notice the difference: when we execute

```
SELECT age_category(25)
```

...it will return the value 'Adult'.

If you execute

```
SELECT age_category_dyn('age')
```

...it will return a text line that contains the part of code

```
CASE
    WHEN age <= 2 THEN 'Infant'
    WHEN age<= 12 THEN 'Child'
    WHEN age< 65 THEN 'Adult'
    ELSE 'Senior'
END
```

To use this function, you will need to package the SELECT statement into a function, but we already know how to do that—see Listing 13-10.

Listing 13-10. Using the `age_category_dyn` function to build a dynamic SQL query

```
CREATE TYPE passenger_age_cat_record AS (
    passenger_id int,
    age_category text
);
CREATE OR REPLACE FUNCTION passenger_age_category_select ()
RETURNS setof passenger_age_cat_record
AS
$body$
BEGIN
    RETURN QUERY
    EXECUTE $$SELECT
        passenger_id,
        $$||age_category_dyn('age')||$$ AS age_category
    FROM passenger $$
;
END;
$body$ LANGUAGE plpgsql;
```


Now, we can execute the following statement:

```
SELECT * FROM passenger_age_category_select ()
```

This will take about five seconds to execute, which is more than a statement without any function calls, but still less than when we choose to execute the original version of the `age_category` function. And once again, when we are dealing with real analytical queries, the effect will be more visible. In fact, using a larger table (e.g., by building the `passenger_large` table, similarly to how `boarding_pass_large` was built in Chapter 7), we can observe similar behavior with a more pronounced difference in execution time.

Some might argue that going to the trouble of creating functions that generate code is not worth the performance gains. To reiterate, there is no universal principle for whether or not creating functions is beneficial—either for performance, code factoring, or portability. Chapter 11 mentioned that code factoring does not work for PG/PL SQL functions the way it works for object-oriented programming languages and promised to provide some examples. This section gives one of those examples. Here, the function `age_category_dyn` helps code factoring, because updates to the age category assignment must be made in only one place. At the same time, it has less impact on performance than a more traditional function with parameters. Most of the time, building a function that executes dynamic SQL takes some time in the beginning, because debugging is more difficult. However, when the function is already in place, it takes little time to make changes. Deciding which time is more critical—the initial development time or average execution time—can only be done by application and/or database developers.

Using Dynamic SQL for Flexibility

The technique described here is most commonly used in OLTP systems, although, once again, it's not strictly limited to one type of environment.

Often, a system allows a user to select an arbitrary list of search criteria, perhaps using some drop-down lists or other graphical means to construct a query.

The user does not (and should not) know anything about the way the data is stored in the database. However, the search fields may be located in different tables, search criteria may have different selectivity, and, in general, the `SELECT` statement may look very different depending on the selection criteria.

Let's look at an example. Suppose a function is needed to search for a booking using any combination of the following values:

- Email (or the beginning portion of email)
- Departure airport
- Arrival airport
- Departure date
- Flight ID

Is there any way to implement this function efficiently without defaulting to Elasticsearch?!

A common approach to this problem is something like Listing 13-11.

Listing 13-11. Function supporting search using different combinations of parameters

```
CREATE TYPE booking_record_basic AS(
    booking_id bigint,
    booking_ref text,
    booking_name text ,
    account_id integer,
    email text );
CREATE OR REPLACE FUNCTION select_booking (
    p_email text,
    p_dep_airport text,
    p_arr_airport text,
    p_dep_date date,
    p_flight_id int)
RETURNS SETOF booking_record_basic
AS
$func$
BEGIN
RETURN QUERY
SELECT DISTINCT
    b.booking_id,
    b.booking_ref,
```

```

        booking_name,
        account_id,
        email
FROM booking b
JOIN booking_leg bl USING (booking_id)
JOIN flight f USING (flight_id)
WHERE (p_email IS NULL OR lower(email) LIKE p_email||'%)
AND (p_dep_airport IS NULL OR departure_airport=p_dep_airport)
AND (p_arr_airport IS NULL OR arrival_airport=p_arr_airport)
AND (p_flight_id IS NULL OR bl.flight_id=p_flight_id);
END;
$func$ LANGUAGE plpgsql;

```

This function will always return the correct result, but from a performance standpoint, its behavior will be, at minimum, difficult to predict. Note that when searching by email address, the joins to the `booking_leg` and `flight` tables are not needed, but they will be still present.

In the following, we recoded one possible sequence of calls and the execution times we observed for these executions.

#1. Search on email.

```

SELECT DISTINCT
    b.booking_id,
    b.booking_ref,
    b.booking_name,
    b.email
FROM booking b
WHERE lower(email) like 'lawton52%'

```

As a `SELECT`, this takes 2.1 seconds.

```

SELECT * FROM select_booking ('lawton52',
NULL,
NULL,
NULL,
NULL
)

```

A comparable function execution takes 3.5 seconds.

#2. Filter on email and flight_id.

```
SELECT DISTINCT
    b.booking_id,
    b.booking_ref,
    b.booking_name,
    b.email
FROM booking b
JOIN booking_leg bl USING (booking_id)
WHERE lower(email) like 'lawton52%'
AND flight_id= 2605
```

The SELECT takes 80 ms.

```
SELECT * FROM select_booking (
    'lawton52',
    NULL,
    NULL,
    NULL,
    2605
)
```

Function execution takes the same 80 ms.

#3. Criteria on email, departure airport, and arrival airport.

```
SELECT DISTINCT
    b.booking_id,
    b.booking_ref,
    b.booking_name,
    b.email
FROM booking b
JOIN booking_leg bl USING (booking_id)
JOIN flight f USING (flight_id)
WHERE lower(email) like 'lawton52%'
AND departure_airport='ORD'
AND arrival_airport='JFK'
```

The SELECT takes 80 ms.

```
SELECT * FROM select_booking (
    'lawton52',
    'ORD',
    'JFK',
    NULL,
    NULL
)
```

Function execution with the same parameters takes 200 ms.

#4. Criteria on email, departure airport, arrival airport, and scheduled departure.

```
SELECT DISTINCT
    b.booking_id,
    b.booking_ref,
    b.booking_name,
    b.email
FROM booking b
JOIN booking_leg bl USING (booking_id)
JOIN flight f USING (flight_id)
WHERE lower(email) like 'lawton52%'
AND departure_airport='ORD'
AND arrival_airport='JFK'
AND scheduled_departure BETWEEN '07-26-2023' AND '07-27-2023'
```

SELECT takes 46 ms.

```
SELECT * FROM select_booking (
    'lawton52',
    'ORD',
    'JFK',
    '2023-07-26',
    NULL
)
```

Function execution takes 155 ms.

#5. Search on email and scheduled departure.

```
SELECT DISTINCT
    b.booking_id,
    b.booking_ref,
    b.booking_name,
    b.email
FROM booking b
JOIN booking_leg bl USING (booking_id)
JOIN flight f USING (flight_id)
WHERE lower(email) like 'lawton52%'
AND scheduled_departure BETWEEN '07-30-2023' AND '07-31-2023'
```

SELECT takes 1.3 seconds.

```
SELECT * FROM select_booking (
    'lawton52',
    NULL,
    NULL,
    '2023-07-30',
    NULL
)
```

Function execution takes **35** seconds.

#6. Search on flight_id.

```
SELECT DISTINCT
    b.booking_id,
    b.booking_ref,
    b.booking_name,
    b.email
FROM booking b
JOIN booking_leg bl USING (booking_id)
WHERE flight_id= 27191
```

SELECT takes 56 ms.

```

SELECT * FROM select_booking (
    NULL,
    NULL,
    NULL,
    NULL,
    27191
)

```

Function execution takes **10** seconds.

As we discussed previously, execution times for different function invocations could be even longer if the first five function executions in the current session produce an execution plan that is suboptimal for subsequent executions. While experimenting with this function, we managed to find a sequence of function invocations that made the last example run for three minutes. But even a ten-second execution time in a situation where the expected response time is under 100 ms can cause an application timeout. Even if such situations are extremely rare, they are unacceptable in production systems.

Can we find an alternative solution? Like the previous example, it's possible to write a function that builds a SELECT dynamically depending on which parameters are passed. In addition, it benefits from being analyzed before each execution.

The source code for the new function is shown in Listing 13-12.

Listing 13-12. A function that builds dynamic SQL to search by different criteria

```

CREATE OR REPLACE FUNCTION select_booking_dyn (
    p_email text,
    p_dep_airport text,
    p_arr_airport text,
    p_dep_date date,
    p_flight_id int)
returns setof booking_record_basic
AS
$func$
DECLARE
v_sql text:=
'SELECT DISTINCT
    b.booking_id,
    b.booking_ref,

```

```

        booking_name,
        account_id,
        email
FROM   booking b ';
v_where_booking text;
v_where_booking_leg text;
v_where_flight text;
BEGIN
IF p_email IS NOT NULL
    THEN v_where_booking := $$ lower(email) like $$ ||quote_literal(p_
email||'%'); END IF;
IF p_flight_id IS NOT NULL
    THEN v_where_booking_leg:= $$ flight_id=$$||p_flight_id::text;
END IF;
IF p_dep_airport IS NOT NULL
    THEN v_where_flight:=concat_ws($$ AND $$, v_where_flight,
                                     $$departure_airport=$$||
                                     quote_literal
                                     (p_dep_airport));
END IF;
IF p_arr_airport IS NOT NULL
    THEN v_where_flight:=concat_ws($$ AND $$,v_where_flight,
                                     $$arrival_airport=$$||
                                     quote_literal
                                     (p_arr_airport));
END IF;
IF p_dep_date IS NOT NULL
    THEN v_where_flight:=concat_ws($$ AND $$,v_where_flight,
                                     $$scheduled_departure BETWEEN $$||
                                     quote_literal(p_dep_date)
                                     ||$$::date AND
                                     $$||quote_literal(p_dep_date)||
                                     $$::date+1$$);
END IF;

```



```

IF v_where_flight IS NOT NULL OR v_where_booking_leg IS NOT NULL
    THEN v_sql:=v_sql||$$ JOIN booking_leg bl USING (booking_id) $$;
END IF;
IF v_where_flight IS NOT NULL
    THEN v_sql:=v_sql ||$$ JOIN flight f USING (flight_id) $$;
END IF;
v_sql:=v_sql ||$$ WHERE $$||concat_ws($$ AND $$,v_where_booking, v_where_
booking_leg, v_where_flight);
return query EXECUTE (v_sql);
END;
$func$ LANGUAGE plpgsql;

```

This is a lot of code to read! Let's walk through it and review what, exactly, is going on here.

The parameters of the new function are exactly the same as those of the old function, and the result type is also the same, but the function body is completely different. At a high level, this function builds a statement to be executed later in the `v_sql` text variable.

Building the query dynamically means that we have the option to only include the joins that are needed. The booking table is always needed, which is why the initial value of `v_sql` is set to

```

'SELECT DISTINCT
    b.booking_id,
    b.booking_ref,
    booking_name,
    account_id,
    email
FROM booking b ';

```

Then, depending on what other parameters are passed as NOT NULL, the function determines which other tables are needed. It may only be the table `booking_leg`. This is the case if the `p_flight_id` parameter is null, that is, flight-related parameters are not used. Alternately, it might be both tables: `booking_leg` and `flight`.

After adding all necessary tables, the full search criteria are built by concatenating all conditions with the separator 'AND'. With the search criteria, the `v_sql` statement is finalized and executed. To see what the final query is for different invocations of the function, uncomment the `RAISE NOTICE` statement.

Is this too much work for a performance improvement? Try to compile this function and execute it with the same parameters from the preceding examples. It'll become clear quickly that the `select_booking_dyn()` function execution times do not exceed the execution times of the corresponding SQL statements for every set of parameters—this is especially evident in cases #5 and #6. Moreover, the execution time is predictable and does not depend on the first execution in the current session.

Once again, dynamic functions are not easy to debug, and you may need to include a lot of debugging printouts, but if performance in your production system is critical, the results are well worth the effort.

Using Dynamic SQL to Aid the Optimizer

Since this whole chapter is dedicated to ways to improve query performance by using dynamic SQL, this section header might be puzzling. However, this section concerns a different cause of performance issues. In these examples, dynamic SQL is not used to construct case-specific SQL, but to nudge the optimizer to choose a better execution plan.

Looking closely at all the examples in the previous section, one combination of the search criteria is performing notably poorly, even though the result set is small: the case when the search is on the email on the booking and the departure airport. Even in cases when email is restrictive enough, the optimizer fails to use the index on `booking_id` in the second join. If we execute the query in Listing 13-13, the execution plan shows hash joins—see Figure 13-4.

Listing 13-13. Selecting booking by email and departure airport

```
SELECT DISTINCT
    b.booking_id,
    b.booking_ref,
    b.booking_name,
    b.account_id,
    b.email
FROM booking b
JOIN booking_leg bl USING (booking_id)
JOIN flight f USING (flight_id)
WHERE lower(email) like 'lawton510%'
      AND departure_airport='JFK'
```

The execution time for this query is about four seconds, and the result contains only 16 rows, so this is a small query, and the execution time should be faster.

The reason for this suboptimal plan has been mentioned before—the PostgreSQL optimizer does not estimate the size of intermediate result sets correctly. The actual number of rows filtered by the pattern index is 3941, while the estimate in the plan is 28219.

	QUERY PLAN	
	text	
1	Unique (cost=427531.98..427549.21 rows=1378 width=64) (actual time=1513.733..1513.745 rows=16 loops=1)	
2	-> Sort (cost=427531.98..427535.43 rows=1378 width=64) (actual time=1513.732..1513.739 rows=25 loops=1)	
3	Sort Key: b.booking_id, b.booking_ref, b.booking_name, b.email	
4	Sort Method: quicksort Memory: 27kB	
5	-> Hash Join (cost=69747.98..427460.13 rows=1378 width=64) (actual time=237.830..1513.670 rows=25 loops=1)	
6	Hash Cond: (bl.flight_id = f.flight_id)	
7	-> Hash Join (cost=60346.96..417824.26 rows=89468 width=68) (actual time=9.850..1365.625 rows=665 loops=1)	
8	Hash Cond: (bl.booking_id = b.booking_id)	
9	-> Seq Scan on booking_leg bl (cost=0.00..310506.66 rows=17893566 width=8) (actual time=0.314..734.995 rows=17893566 loops=1)	
10	-> Hash (cost=59994.26..59994.26 rows=28216 width=64) (actual time=8.011..8.015 rows=222 loops=1)	
11	Buckets: 32768 Batches: 1 Memory Usage: 277kB	
12	-> Bitmap Heap Scan on booking b (cost=489.65..59994.26 rows=28216 width=64) (actual time=0.059..7.953 rows=222 loops=1)	
13	Filter: (lower(email) ~~ 'lawton510%':text)	
14	Heap Blocks: exact=215	
15	-> Bitmap Index Scan on booking_email_lower_pattern (cost=0.00..482.59 rows=28216 width=0) (actual time=0.034..0.034 rows=222 loops=1)	
16	Index Cond: ((lower(email) ~>= 'lawton510':text) AND (lower(email) ~<= 'lawton511':text))	
17	-> Hash (cost=9269.50..9269.50 rows=10521 width=4) (actual time=147.605..147.605 rows=10530 loops=1)	
18	Buckets: 16384 Batches: 1 Memory Usage: 499kB	
19	-> Bitmap Heap Scan on flight f (cost=121.96..9269.50 rows=10521 width=4) (actual time=2.908..146.103 rows=10530 loops=1)	
20	Recheck Cond: (departure_airport = 'JFK':bpchar)	
21	Heap Blocks: exact=4085	
22	-> Bitmap Index Scan on flight_departure_airport (cost=0.00..119.33 rows=10521 width=0) (actual time=1.597..1.597 rows=10530 loops=1)	
23	Index Cond: (departure_airport = 'JFK':bpchar)	

Figure 13-4. Execution plan for Listing 13-13 with hash joins

The technique to optimize this query is literally to help the optimizer do its job and remove the need to estimate the size of the result set. How? First, find the booking IDs that correspond to the email address that is being searched for, and then pass the list of booking_ids to the main SELECT statement. Note: The function that we use to illustrate this case is very case-specific and used for illustrative purposes only (Listing 13-14). A function with a more generalized approach closer to what would be used in a production system would be massive.

Listing 13-14. Dynamic SQL to improve the code from Listing 13-13

```

CREATE OR REPLACE FUNCTION select_booking_email_departure(
    p_email text,
    p_dep_airport text)
RETURNS SETOF booking_record_basic AS
$body$
DECLARE
    v_sql text;
    v_booking_ids text;
BEGIN
    EXECUTE $$SELECT array_to_string(array_agg(booking_id), ',')
                FROM booking
                WHERE lower(email) like $$||quote_literal(p_email)||'%'
    INTO v_booking_ids;
    v_sql=
    $$SELECT DISTINCT
        b.booking_id,
        b.booking_ref,
        b.booking_name,
        b.account_id,
        b.email
    FROM booking b
    JOIN booking_leg bl USING(booking_id)
    JOIN flight f USING (flight_id)
    WHERE b.booking_id IN ($$||v_booking_ids||$$)
        AND departure_airport=$$||quote_literal(p_dep_airport);
    RETURN QUERY EXECUTE v_sql;
END;
$body$ LANGUAGE plpgsql;

```

Why does this work? We know that the search by email is going to be relatively restrictive, because what is passed is nearly the whole email address or, at least, the essential part of it. So, in the first step, the relatively small number of bookings with this email is preselected and saved in the text variable `v_booking_ids`. Then, the `SELECT` is constructed with an explicit list of `booking_ids`.

Executing this new function

```
SELECT * FROM select_booking_email_departure('lawton510','JFK')
```

...the execution time will be just 200 ms. Examine the EXPLAIN command output for the generated SQL, including the execution plan as it appears in Figure 13-5.

	QUERY PLAN text
1	Unique (cost=14591.09..14591.25 rows=11 width=68) (actual time=5.429..5.442 rows=16 loops=1)
2	-> Sort (cost=14591.09..14591.11 rows=11 width=68) (actual time=5.428..5.431 rows=25 loops=1)
3	Sort Key: b.booking_id, b.booking_ref, b.booking_name, b.account_id, b.email
4	Sort Method: quicksort Memory: 27kB
5	-> Nested Loop (cost=1.29..14590.89 rows=11 width=68) (actual time=0.656..5.398 rows=25 loops=1)
6	-> Nested Loop (cost=0.87..14275.97 rows=704 width=72) (actual time=0.051..2.419 rows=665 loops=1)
7	-> Index Scan using booking_pkey on booking b (cost=0.43..1845.30 rows=222 width=68) (actual time=0.039..1.014 rows=222 loops=1)
8	Index Cond: (booking_id = ANY ({3613355,957543,1111315,1253518,628907,629954,5183078,5184235,5376394,5406828,5454800,5582121,7213
9	-> Index Scan using booking_leg_booking_id on booking_leg bl (cost=0.44..55.86 rows=13 width=8) (actual time=0.005..0.006 rows=3 loops=222)
10	Index Cond: (booking_id = b.booking_id)
11	-> Index Scan using flight_pkey on flight f (cost=0.42..0.45 rows=1 width=4) (actual time=0.004..0.004 rows=0 loops=665)
12	Index Cond: (flight_id = bl.flight_id)
13	Filter: (departure_airport = 'JFK'::bpchar)
14	Rows Removed by Filter: 1
15	Planning Time: 0.816 ms
16	Execution Time: 5.483 ms

Figure 13-5. Execution plan for dynamic SQL with the list of booking_ids

Even with several thousand IDs, the index-based access proves more efficient.

FDWs and Dynamic SQL

As mentioned in the Introduction, detailed discussion of distributed queries is out of the scope of this book. However, since dynamic SQL is covered, this is a good opportunity to make a few remarks about working with *foreign data wrappers* (FDWs).

A foreign data wrapper is a library that can communicate with an external data source (i.e., data that resides outside your PostgreSQL server), hiding the details of connecting to the data source and obtaining data from it.

An FDW is a very powerful tool, and more and more foreign data wrappers for different types of databases are becoming available. PostgreSQL does an outstanding job optimizing queries that include *foreign tables*, that is, mappings of the tables from the external systems. However, since the access to external statistics may be limited, especially when the external systems are not PostgreSQL based, the optimization may be not so precise. We have found it very helpful to use the techniques described in the previous section.

The first way to optimize would be to run the local part of the query identifying which records are needed from the remote server and then access a remote table. An alternative way is to send a query with constant-defined conditions (e.g., `WHERE update_ts > CURRENT_DATE - 3`) to the remote site, pull the remote data to the local site, and then execute the rest of the query. Using one of these two techniques helps minimize inconsistencies in the execution time.

Summary

Dynamic SQL is an exceptionally powerful tool in PostgreSQL, which is not utilized enough by database developers. Using dynamic SQL can improve performance in situations where all other optimization techniques fail.

Dynamic SQL works best within functions; a SQL statement is generated based on function input parameters and then executed. It can be used both in OLTP and OLAP environments.

If you choose to use dynamic SQL for your project, be ready for extensive and time-consuming debugging. It might feel discouraging in the beginning, but the performance improvements are well worth it.

CHAPTER 14

Avoiding the Pitfalls of Object-Relational Mapping

Chapter 11 discussed a typical interaction between an application and a database and explained ORIM (object-relational impedance mismatch) and how it affects performance. It also stated that any potential solution should allow operating with large objects (i.e., datasets) and should support the exchange of complex objects. This chapter introduces an approach developed by us and successfully utilized in a production environment. This approach is called NORM (No-ORM).

We are by no means pioneers in the quest to overcome object-relational impedance mismatch, nor are we the first to propose an alternative to ORM. NORM is only one of many possible solutions. However, one feature that makes NORM stand out among other tools is the ease of use by application developers.

The NORM GitHub repo (<https://github.com/hettie-d/NORM>) contains some documentation on the approach and an example of the code built according to the NORM methodology. Most importantly, it contains the set of procedures that allow to automate the function code generation. More details will be provided later in this chapter.

Why Application Developers Like NORM

Often, new development methodologies require application developers to make significant changes to the development process, which inevitably leads to lower productivity. It is not unusual for potential performance gains to fail to justify the increase in development time. After all, developer time is the most expensive resource in any project.

In Chapter 12, the benefits of using functions were preceded by the caveat “if you can convince application developers.” And often, you can’t convince them, because of the difficulties of adapting to a new programming style. That is not the case with NORM. In the following sections, we will explain the appeal of this approach for both application developers and database developers.

ORM vs. NORM

Chapter 11 discussed a bottleneck in data exchange created by ORM. Figure 14-1 is a copy of Figure 11-1 from Chapter 11, and it represents the dataflow between an application and a database.

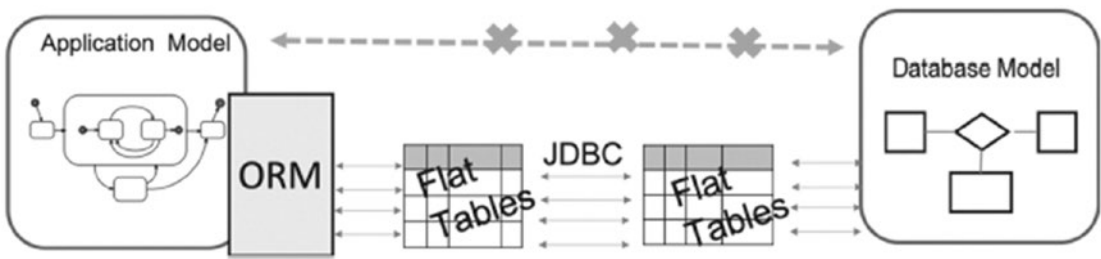


Figure 14-1. *How ORM works*

The major problem is that complex objects from the application model are disassembled into atomic objects before communicating with the database, generating too many too small queries, which bring down system performance. An even more important problem is that the correspondence between application complex objects and database complex data structures is lost.

The approach proposed by NORM is presented in Figure 14-2.

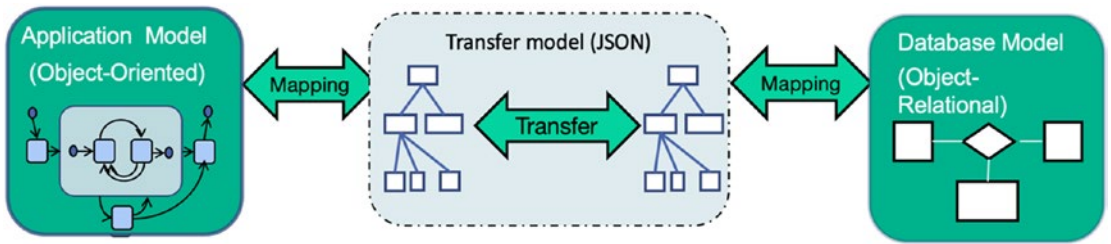


Figure 14-2. *How NORM works*

In this figure, A-Model is an application model, D-Model is a database model, and T-Model is a *transfer model*. The presence of the T-Model is a unique feature of the NORM approach, which makes the mapping *symmetrical*. We are not trying to build a database model based on an application model, nor are we insisting on creating database objects first.

Instead, we call for *a contract* to be established between the application layer and the database, like the way you would see a contract over a RESTful web service. The NORM contract requires that an application controller performs at most two accesses to the database: one to obtain all data needed for the function and another to send request for data modification. Since the whole complex hierarchical object is sent or received as a minimal unit of transfer, it is extremely important to properly define it for each case (or choose the right class/hierarchy from the ones already defined). Different implementations of NORM may use different means to represent hierarchies. Our implementation uses one of the most popular notations, namely, JSON.

The contract, or a T-Model, comes in a form of a JSON schema describing the structure of complex objects, their relationships in a hierarchy, and mapping of these objects to the database objects (tables and columns).

Through this contract, it is possible to simplify the persistence of objects by serializing the objects into JSON payloads that the database can consume. This results in one database call to persist an object regardless of its structure or complexity.

Likewise, when retrieving objects, the application can deserialize the result coming back from the database to a model in a single database call. It can also pass additional parameters as a part of the contract to tell the database that it needs additional pieces of the model, similar to an ODATA web service request.

Many application developers love the simplified implementation of the data access layer on the application side. The fact that NORM uses a contract to determine the inputs and outputs of every call to the database allows application developers to code to the

contract and easily mock out any dependencies when testing, as the calls to and from the database will abide by the contract. Thus, after a contract is established, database and application developers can do their part simultaneously and independently from each other. Moreover, different groups of application developers can use the same contract for different projects.

On the application side, almost all modern object-oriented languages have libraries for serializing and deserializing objects. As each new database interaction occurs, it is possible to reuse the same pattern for implementation.

This allows application developers to spend more time designing the JSON payload to ensure it meets the current and future needs of the business. Reusing the same pattern of interactions also reduces implementation time, minimizes the likelihood of defects, and allows minimal code changes to impact the entire database access implementation. More importantly, an application usually needs different complex objects and thus requires multiple contracts, although these complex objects can be built from the same database.

NORM Explained

To illustrate how NORM works, let's get back to the example in Chapter 11.

Figure 14-3 represents a subset of the `postgres_air` Entity-Relationship diagram used to build the example.

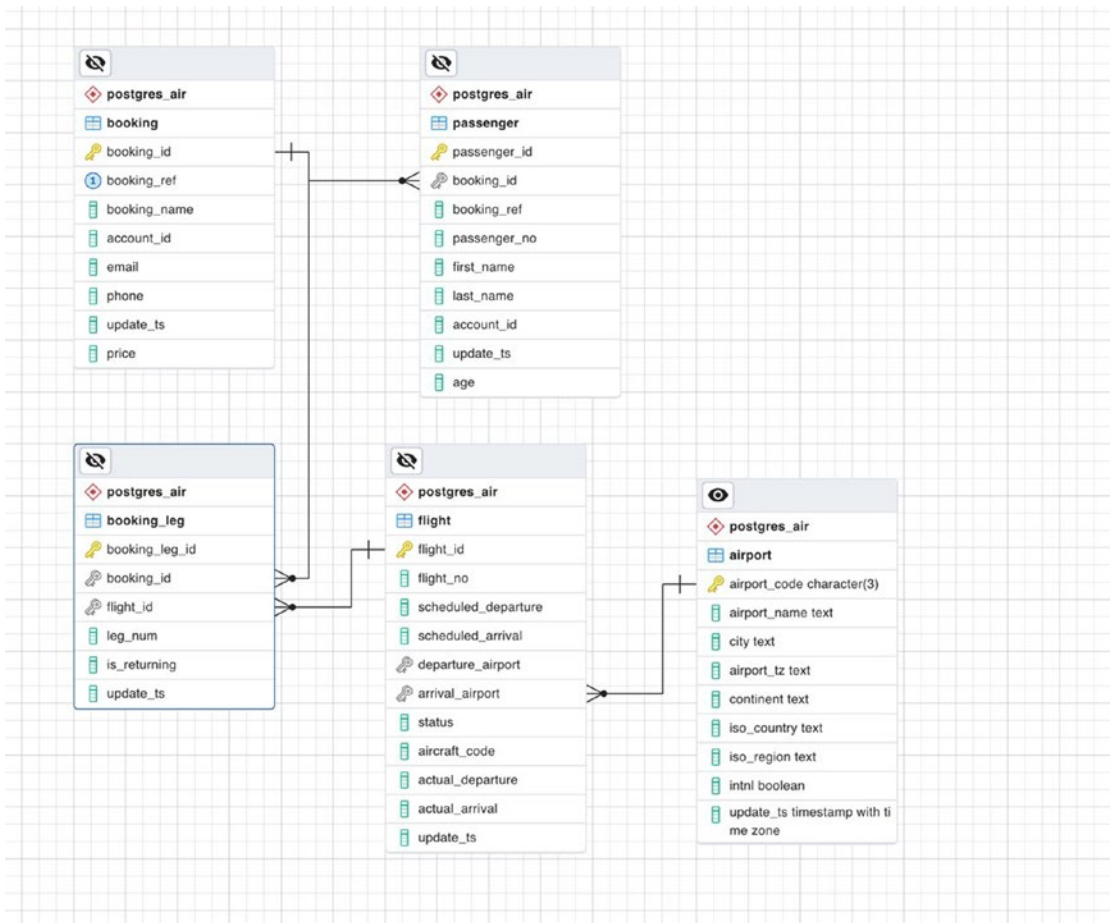


Figure 14-3. ERD for the case study

In Chapter 11, discussing the interaction between the application and the database, we drafted an object (which we can now call a T-object hierarchy) that represented all the information related to a booking. From the airline passenger perspective, a booking represents their travel itinerary. To keep the code sample readable, we eliminated some details that can usually be found on an itinerary. Specifically, we include airport codes, but not cities where airports are located, avoiding the need to include mapping of the airport table.

The ERD in Figure 14-3 presents all tables and relationships needed to build the mapping from a database object to a transfer object hierarchy. The relationships of transfer objects in the hierarchy are presented in Figure 14-4.

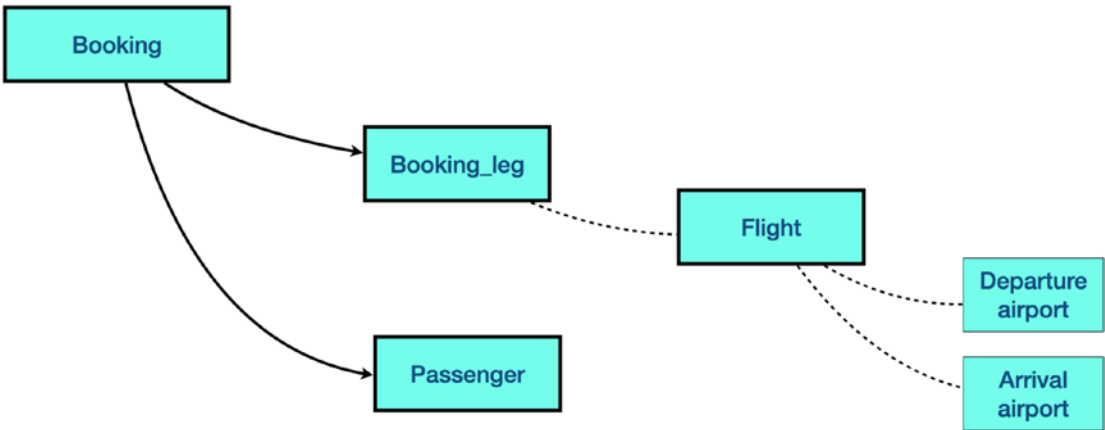


Figure 14-4. *The relationships in the transport object hierarchy (contract)*

The contract described as JSON is shown in Listing 14-1. This listing contains a complete description of the booking hierarchy.

Object definitions are placed under the “definitions” key. All object definitions are on the same level. The hierarchical structure is specified with references placed in “\$ref” keys. Line 2 requires that the unit of transfer is an array of JSON objects. Lines 5–7 specify that the root object of the hierarchy has type booking.

The definition of this object contains keys “passengers” and “booking_legs” that are arrays of objects “passenger” and “booking_leg,” respectively (lines 15–25).

Note that both database tables passenger and booking_leg contain a foreign key column booking_id. This field is not included to JSON as the relationships are represented with hierarchical structure of JSON.

Another method to build a hierarchy is used in the definition of “booking_leg.” This object contains a single embedded object of type “flight.” The important difference is that field flight_id is contained in the booking_leg object.

The flight data cannot be updated through this hierarchy (as such update would affect other bookings on the same flight).

Instead, the booking_leg.flight_id (lines 88–90) can be modified if rebooking is needed.

Similarly, the flight JSON object contains name and city of departure and arrival airports. These fields cannot be updated through this hierarchy because changes in the booking cannot affect properties of an airport.

Listing 14-1. The contract in the form of a JSON schema

```

1  {
2      "type": "array",
3      "title": "booking_hierarchy",
4      "description": "all booking details",
5      "items": {
6          "$ref": "#/definitions/booking"
7      },
8      "definitions": {
9          "booking": {
10             "type": "object",
11             "properties": {
12                 "booking_id": {
13                     "type": "number"
14                 },
15                 "passengers": {
16                     "type": "array",
17                     "items": {
18                         "$ref": "#/definitions/passenger"
19                     }
20                 },
21                 "booking_legs": {
22                     "type": "array",
23                     "items": {
24                         "$ref": "#/definitions/booking_leg"
25                     }
26                 },
27                 "booking_reference": {
28                     "type": "string"
29                 }
30             }
31         },
32         "flight": {
33             "type": "object",
34             "properties": {

```

```

35         "flight_no": {
36             "type": "string"
37         },
38         "arrival_city": {
39             "type": "string"
40         },
41         "departure_city": {
42             "type": "string"
43         },
44         "scheduled_arrival": {
45             "type": "string"
46         },
47         "scheduled_departure": {
48             "type": "string"
49         },
50         "arrival_airport_code": {
51             "type": "string"
52         },
53         "arrival_airport_name": {
54             "type": "string"
55         },
56         "departure_airport_code": {
57             "type": "string"
58         },
59         "departure_airport_name": {
60             "type": "string"
61         }
62     }
63 },
64 "passenger": {
65     "type": "object",
66     "properties": {
67         "last_name": {
68             "type": "string"
69         }
70     },

```

```

71         "account_id": {
72             "type": "number"
73         },
74         "first_name": {
75             "type": "string"
76         },
77         "passenger_id": {
78             "type": "number"
79         },
80         "passenger_no": {
81             "type": "number"
82         }
83     },
84 },
85 "booking_leg": {
86     "type": "object",
87     "properties": {
88         "flight_id": {
89             "type": "number"
90         },
91         "flight": {
92             "type": "object",
93             "items": {
94                 "$ref": "#/definitions/flight"
95             }
96         },
97         "leg_num": {
98             "type": "number"
99         },
100         "booking_leg_id": {
101             "type": "number"
102         }
103     }
104 },
105 }
106 }
```

Note that this schema represents the contract, that is, the object structure that the application expects to receive. It differs significantly from how the data is stored in the database, and the most important part is that the database implementation has no impact on how the application interacts with it, as long as the database response remains in accordance with the contract.

An example JSON object following this contract is shown in Figure 14-5.

```

1  {
2    "booking_id": 10679,
3    "passengers": [
4      {
5        "last_name": "CHARLES",
6        "account_id": 193770,
7        "first_name": "ARYA",
8        "passenger_id": 34482,
9        "passenger_no": null
10     }
11   ],
12   "booking_legs": [
13     {
14       "flight": {
15         "flight_no": "36",
16         "arrival_city": "FRANKFURT",
17         "departure_city": "ADDIS ABABA",
18         "scheduled_arrival": "2023-06-03T12:10:00-05:00",
19         "scheduled_departure": "2023-06-03T05:15:00-05:00",
20         "arrival_airport_code": "FRA",
21         "arrival_airport_name": "Frankfurt am Main Airport",
22         "departure_airport_code": "ADD",
23         "departure_airport_name": "Addis Ababa Bole International Airport"
24       },
25       "leg_num": 1,
26       "flight_id": 6392,
27       "booking_leg_id": 16734488
28     },
29     {
30       "flight": {
31         "flight_no": "37",
32         "arrival_city": "ADDIS ABABA",
33         "departure_city": "FRANKFURT",
34         "scheduled_arrival": "2023-06-10T21:00:00-05:00",
35         "scheduled_departure": "2023-06-10T14:05:00-05:00",
36         "arrival_airport_code": "ADD",
37         "arrival_airport_name": "Addis Ababa Bole International Airport",
38         "departure_airport_code": "FRA",
39         "departure_airport_name": "Frankfurt am Main Airport"
40       },
41       "leg_num": 2,
42       "flight_id": 32604,
43       "booking_leg_id": 16734489
44     }
45   ],
46   "booking_reference": "EN6B12"
47 }

```

Figure 14-5. *Transfer object as JSON*

In short, the interaction between the application and the database can be summarized as follows:

1. The application serializes data into JSON format and sends it to the database by calling a corresponding database function.

2. A database function parses the JSON that was passed as a parameter and executes whatever the function is supposed to do: either a search or data transformation. PostgreSQL has a very well-developed JSON support with a rich set of functions, which makes it easy to manipulate JSON on the database side.
3. The result set is converted to JSON and passed to the application, where it is deserialized and is ready to be consumed by the application.

NORM in the Application Perspective

On the application side, the Python classes in Listing 14-2 are mapped to the same transfer object.

Listing 14-2. Postgres Air application classes

```
from typing import List
from pydantic import BaseModel, Field
from datetime.datetime import datetime, timezone
from phonenumbers import PhoneNumber

class Booking(BaseModel):
    booking_id: int
    booking_ref: str
    booking_legs: List[BookingLeg]
    booking_name: str
    account_id: int
    email: str
    phone: PhoneNumber
    update_ts: datetime
    price: decimal

class BookingLeg(BaseModel):
    booking_leg_id: int
    flight: Flight
    is_returning: bool
    update_ts: datetime
```

```

class Flight(BaseModel):
    flight_id: int
    flight_no: int
    scheduled_departure: datetime
    scehduled_arrival: datetimee
    departure_airport: Airport
    arrival_airport: Airport
    status: str
    aircraft_code: str
    actual_departure: datetime
    actual_arrival: datetime
    update_ts=datetime

class Airport(BaseModel):
    airport_code: str = Field(min_length=3,max_length=3)
    airport_name: str
    airport_tz: timezone
    city: str
    continent: str
    iso_country: str
    iso_region: str
    intl: bool
    update_ts: datetime

class Passenger(BaseModel):
    passenger_id: int
    booking: Booking
    passenger_no: int
    first_name: str
    last_name: str
    account_id: int
    update_ts: datetime
    age: int

```

It is worth mentioning that we can build completely different transfer objects using the same set of tables. For example, before any flight departs, a document that is called manifest has to be produced. This document lists all the passengers on the flight along with their seat assignments. The transfer object for the manifest is presented in Figure 14-6.

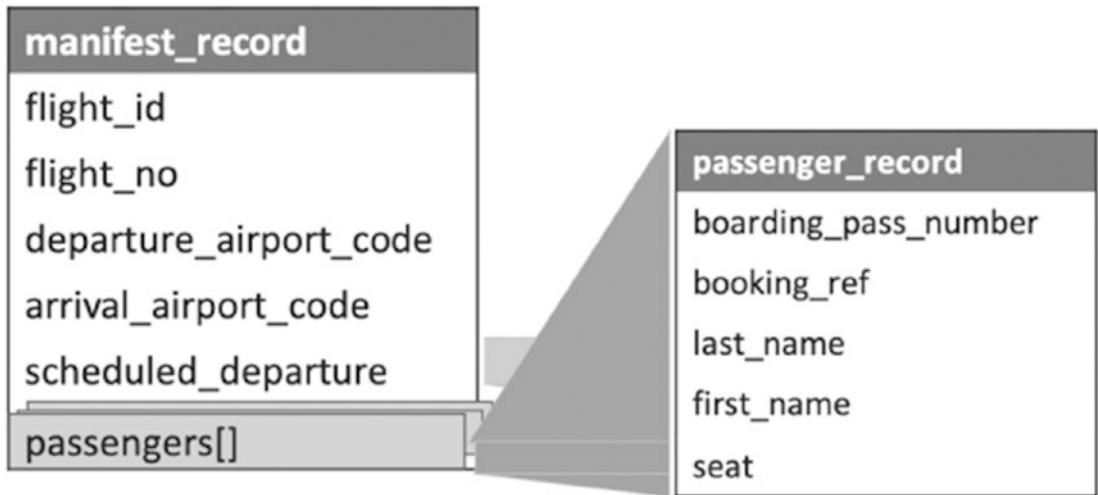


Figure 14-6. Transfer object for the flight manifest

The matching JSON is shown in Figure 14-7.

```
{
  "flight": {
    "flight_id": 13650,
    "flight_no": "1245",
    "scheduled_arrival": "2020-06-05T02:45:00-05:00",
    "scheduled_departure": "2020-06-05T00:20:00-05:00",
    "arrival_airport_code": "CDG",
    "arrival_airport_name": "Charles de Gaulle International Airport",
    "departure_airport_code": "JFK",
    "departure_airport_name": "John F Kennedy International Airport"
  },
  "boarding_passes": [
    {
      "seat": "1E",
      "flight_no": "1245",
      "last_name": "LEWIS",
      "first_name": "ELIA",
      "boarding_time": "2020-06-04T23:50:00-05:00",
      "booking_leg_id": 17564910,
      "arrival_airport": "CDG",
      "boarding_pass_id": 1247796,
      "departure_airport": "JFK"
    },
    {
      "seat": "1F",
      "flight_no": "1245",
      "last_name": "LEVY",
      "first_name": "ALEXANDER",
      "boarding_time": "2020-06-04T23:50:00-05:00",
      "booking_leg_id": 17564910,
      "arrival_airport": "CDG",
      "boarding_pass_id": 1247797,
      "departure_airport": "JFK"
    }
  ],
  ...
}
```

Figure 14-7. Manifest object as JSON

NORM from a Database Perspective

In the early stages of the NORM project, the only serious objection to its adoption was the necessity of manually creating all data manipulation functions. Agile teams didn't want to spend time on writing additional code, especially when ORMs generate code with similar functionality (almost) without human interaction. Nor did they want to have an additional person (a database developer) on the team. To address these concerns, we added an auto-generation module to NORM, which we call NORM-GEN. In this section, we will explain how to generate types and functions on the database side, instead of

creating them manually. To achieve this goal, we need to make one important change to our contract, that is, to find a way to define the mapping between application classes and database objects.

Mapping JSON to the Database

To enable the automatic generation of the database objects, we extend the JSON schema syntax with a `db_mapping` key that can be specified for the whole hierarchy, an object, or a key. The mapping specifies the database schema, table names and column names corresponding to the JSON object and keys, as well as primary and foreign key columns that define relationships between objects.

Listing 14-3 shows how the mapping is specified for a part of the contract, namely, for the passenger object. Lines 4–7 of this listing define the table corresponding to the JSON object, the primary key of the table, the primary key of the parent object, and the name of the complex data type that represents the object in the database before conversion to JSON. The explicit mapping of lower-level objects can be omitted when JSON key names coincide with the table column names.

Listing 14-3. Database mapping for the passenger object

```

1      "passenger": {
2          "type": "object",
3          "db_mapping": {
4              "pk_col": "passenger_id",
5              "db_table": "passenger",
6              "record_type": "passenger_record",
7              "parent_fk_col": "booking_id"
8          },
9          "properties": {
10             "last_name": {
11                 "type": "text"
12             },
13             "account_id": {
14                 "type": "number"
15             },
16             "first_name": {
17                 "type": "text"

```

```

18         },
19         "passenger_id": {
20             "type": "number"
21         },
22         "passenger_no": {
23             "type": "number"
24         }
25     }
26 },

```

Generating Database Code

The contract is used to generate

- Database type definitions that represent the hierarchy in the form of nested complex objects and arrays.
- A function that selects full hierarchies from the database based on a list of primary keys of the root objects. The hierarchy is built as a database complex nested object, which is then converted into JSON format.
- A function returning a list of root object keys based on a search request specified as an input parameter.
- A function that performs modifications of specified objects or their sub-objects in the hierarchy. Multiple objects and/or sub-objects can be inserted, updated, or deleted in the same request.

Some of the interfaces that applications use to access the database, including JDBC, do not support JSON as a data type. To bypass this, the JSON is converted into a list of text strings right before the transfer and converted back to JSON immediately after transfer.

Now, let's get more specific and show how this is achieved.

Listing 14-4 shows generated type definitions for the booking hierarchy. These definitions are similar to definitions from Listings 12-17 and 12-20.

We define the types `passenger_record`, `flight_record`, and `booking_leg_record`, which includes a `flight_record` sub-object, and then `booking_record`, which has arrays of these types as components.

Listing 14-4. Booking type definitions

```

/* entering : booking */
/* entering : passenger */
drop type if exists norm.bh_passenger_record cascade;
create type norm.bh_passenger_record as (
    last_name text,
    account_id int4,
    first_name text,
    passenger_id int4,
    passenger_no int4
);
/* entering : booking_leg */
/* entering : flight */
drop type if exists norm.bh_flight_record cascade;
create type norm.bh_flight_record as (
    flight_no text,
    arrival_city text,
    departure_city text,
    scheduled_arrival timestampz,
    scheduled_departure timestampz,
    arrival_airport_code bpchar,
    arrival_airport_name text,
    departure_airport_code bpchar,
    departure_airport_name text
);
drop type if exists norm.bh_booking_leg_record cascade;
create type norm.bh_booking_leg_record as (
    flight_id int4,
    flight norm.bh_flight_record,
    leg_num int4,
    booking_leg_id int4
);
drop type if exists norm.bh_booking_record cascade;
create type norm.bh_booking_record as (
    booking_id int8,

```

```

    passengers norm.bh_passenger_record[],
    booking_legs norm.bh_booking_leg_record[],
    booking_reference text
);

```

Looking at these type definitions, it is clear they do indeed represent the transport object booking from Figure 14-4.

Getting Data from the Database

The contract requires that the application should receive all needed data with the smallest possible number of interactions with the database. Typically, an application should receive a set of complete hierarchies. This implies that the structure of returned data depends on the definition of the hierarchy, but not on the specific query. Listing 14-5 shows the SELECT clause generated for the booking hierarchy.

Listing 14-5. Generated SELECT clause

```

/* selecting booking_hierarchy booking */
select
    array_agg(
        /* Entering booking_record */
        row(top.booking_id ,
            (
                select array_agg( /* Entering passenger_record */
                    row(passengers.last_name ,
                        passengers.account_id ,
                        passengers.first_name ,
                        passengers.passenger_id ,
                        passengers.passenger_no)::norm.bh_passenger_record)
                from postgres_air.passenger passengers
                where top.booking_id = passengers.booking_id
            )
        ),
    (
        select array_agg( /* Entering booking_leg_record */
            row(booking_legs.flight_id ,

```



```

(
select ( /* Entering flight_record */
    row(flight.flight_no ,
        arrival.city ,
        departure.city ,
        flight.scheduled_arrival ,
        flight.scheduled_departure ,
        flight.arrival_airport ,
        arrival.airport_name ,
        flight.departure_airport ,
        departure.airport_name)::norm.bh_flight_record)
from postgres_air.flight flight
    join postgres_air.airport departure on departure.airport_code =
        flight.departure_airport
    join postgres_air.airport arrival on arrival.airport_code = flight.
        arrival_airport
where booking_legs.flight_id = flight.flight_id
)
,
    booking_legs.leg_num ,
    booking_legs.booking_leg_id)::norm.bh_booking_leg_record)
from postgres_air.booking_leg booking_legs
where top.booking_id = booking_legs.booking_id
)
,
    top.booking_ref)::norm.bh_booking_record)
from postgres_air.booking top

```

The SELECT clause shown in Listing 14-5 returns a PostgreSQL array of nested complex objects. This output is converted into JSON format with a single invocation of the `to_json` function.

In contrast with the SELECT clause, the filtering criteria depend on the specific needs of the application and therefore are needed for each query. NORM accepts search conditions in the JSON format that resembles a format found in some JSON-oriented systems. However, NORM provides neither the full power of these languages nor even a

subset of them. Essentially, NORM supports basic comparison operations on any scalar keys in the JSON contact at any level of the hierarchy. An example of search conditions is shown in Listing 14-6.

Listing 14-6. Search conditions in JSON format

```
{
"booking_hierarchy":{
"departure_airport_code":"ORD",
"arrival_city":{"$like":"NEW%"},
"last_name":"Smith"}
}
```

This search will return all bookings that have a passenger with a last name of “Smith” and have a leg with flight from ORD to any city whose name starts with “NEW.” For all these bookings, all passengers and all legs will be included in the result.

Note that search conditions cannot include any join operations because relationships between objects are defined in the hierarchy. If more relationships are needed, another hierarchy must be defined. Listing 14-7 shows the SQL code generated for this search.

Listing 14-7. Generated filtering conditions

```
booking_id IN (
  select booking_id from postgres_air.booking_leg where
    flight_id IN (
      select flight_id from postgres_air.flight where
        arrival_airport IN (
          select airport_code from postgres_air.airport where
            city LIKE ('NEW% '::text) )
        AND departure_airport = ('ORD'::bpchar) ) )
AND booking_id IN (
  select booking_id from postgres_air.passenger where
    last_name = (Smith::text) )
```

This generated code is wrapped in functions that construct a complete query and execute it. Notice that this generated code resembles the code discussed in Chapter 12 and shown in Listing 12-21.

Modifying Data in the Database

NORM can handle any data manipulation operation, that is, INSERT, UPDATE, and DELETE, which are collectively called update requests.

An update request is sent from the application as a complex object and, on the database level, may result in multiple update operations applied to different tables. Once again, database development is contract-driven. A database function receives a JSON object from the application, parses the object, and interprets the actions that are required on the database level.

The NORM approach aims to reduce the number of database operations. Of course, it is not possible to update multiple tables in a single SQL statement, so NORM needs to execute several SQL statements for an update request. However, it collects data from several sub-objects of the update request so that at most one DELETE, one UPDATE, and two INSERT statements are performed for each table that can be modified through the hierarchy, no matter how many hierarchies and sub-objects are modified by the update request.

The generated NORM code relies on the assumption that all tables have a single-column primary key and the values of primary keys are automatically assigned when new rows are inserted.

An update request is sent from the application as a JSON array of hierarchies conforming the contract schema and satisfying a few additional rules:

- If an object at any level of the hierarchy does not include the key mapped to the primary key, this object and all its sub-objects are inserted into the appropriate tables.
- If an object contains the value of the primary key and an additional key “cmd”: “DELETE”, then this object and all its sub-objects are deleted from the database.
- If an object contains a primary key and does not contain the “cmd” key, then the values of other keys of this object are used in the UPDATE operation.

An example of an update request is shown in Listing 14-8. The modifications specified in this request do not necessarily make sense from a business viewpoint; however, they illustrate how modifications are specified.

Listing 14-8. An example of an update request

```
1  [
2      {
3          "booking_id": 556470,
4          "cmd": "DELETE"
5      },
6      {
7          "passengers": [
8              {
9                  "last_name": "Jones",
10                 "account_id": 238648,
11                 "first_name": "Lucy"
12             }
13         ],
14         "booking_legs": [
15             {
16                 "leg_num": 1,
17                 "flight_id": 558238,
18             },
19             {
20                 "leg_num": 2,
21                 "flight_id": 563410
22             }
23         ],
24         "booking_reference": "IYZI42"
25     },
26     {
27         "booking_id": 3974917,
28         "passengers": [
29             {
30                 "cmd": "DELETE",
31                 "passenger_id": 11479596
32             },
33             {
34                 "last_name": "SCOTT",
```

```

35         "first_name": "MILES",
36         "passenger_id": 11479599
37     }
38 ],
39     "booking_legs": [
40         {
41             "flight_id": 432724,
42             "booking_leg_id": 11453272
43         },
44         {
45             "flight_id": 427273,
46             "booking_leg_id": 11453273
47         }
48     ]
49 },
50 {
51     "booking_id": 2733047,
52     "booking_reference": "Q8JX22"
53 }
54 ]

```

The update request shown in Listing 14-8 requests the following actions:

- Delete a whole booking (lines 2–5).
- Insert a new booking (lines 6–25).
- Delete one passenger, update first and last names of another passenger, and change the flight in two booking_leg entries (lines 26–49).
- Update the booking reference in a booking (lines 50–53).

To support update requests, NORM generates several functions that contain SQL DML operations for each table and a function TO_DB that accepts update requests and invokes all the other functions as needed to perform the requested modifications. This function returns a list of primary keys of the top-level objects of all hierarchies that were touched during the execution of the update request.

The generated code is too long to be included in this book, but is available in the NORM repository on GitHub.

Why Not Store JSON?!

At this point, you might wonder why we go into such trouble when PostgreSQL supports the JSON type? Why not store JSON “as is”?

Some reasons were previously discussed in Chapter 9. In particular, Chapter 9 discussed key-value and hierarchical models and explained their limitations. If a booking leg, as defined in this chapter, was stored as a JSON object, flight information would be duplicated, because it belongs to a different hierarchy. Another reason is that JSON is typeless and therefore unreliable in terms of providing a consistent interface to develop against.

In addition, although we can build indexes to facilitate search on specific JSON keys, their performance is worse than B-tree indexes on regular columns. Indexing JSON and related performance concerns are covered in Chapter 15.

Performance Gains

What is the effect of using NORM on performance? As discussed in Chapter 11, this kind of performance difference is difficult to benchmark. We need to measure overall application performance, rather than comparing the speed of separate operations, and the applications themselves in this case are written in very different programming styles. We are not providing any examples of application code in this chapter since it is out of scope of this book.

However, based on our industrial experience, this approach used in place of traditional ORM can improve the performance of application controllers by 10–50 *times*. Moreover, application performance appears to be more consistent, since it avoids the *N + 1 problem* (i.e., when the code needs to load the children of a parent-child relationship: most ORMs have lazy-loading enabled by default, so queries are issued for the parent record and then one query for *each* child record).

Working Together with Application Developers

As discussed many times, overall system performance is not limited to database performance, and optimization starts with gathering requirements. NORM is a very good illustration for this statement.

With NORM, development starts from defining a contract, which allows application and database developers to work in parallel on their tasks. In addition, this contract means that future performance improvements on the database side can be made without making any changes to the application.

Summary

NORM is an approach to application design and development, which allows seamless data exchange between a back end and a data layer eliminating the need for ORM. Applied consistently, it helps produce performant systems while simplifying application development.

NORM is one of several potential solutions; however, it has a proven record of success and can be used as a template for those who want to avoid the potential pitfalls of ORM.

CHAPTER 15

More Complex Filtering and Search

Previous chapters discussed several ways to support filtering and search with indexes in PostgreSQL. Why are more needed, and why haven't these indexes been covered yet? Prior discussion of this topic focused on the most common indexes, those that are needed in nearly any application. However, there are data types that cannot be efficiently supported with indexes such as B-trees.

Full Text Search

Everything discussed in previous chapters is applicable to structured data, and all queries considered so far are Boolean. That is, a row is either needed for computation of the result or not, and a computed row either belongs to the output or not. Nothing resides in between. SQL is a powerful language for structured data and lends itself well to this sort of analysis.

In this section, we consider unstructured data. The most common example of unstructured data is text written in natural language. This kind of text is usually called a *document*. In contrast with structured data, a search for documents is always imprecise, because we are typically interested in a document meaning that is not precisely expressed in the content of the document. However, the criteria must be precisely expressed in a query. Welcome to the world of uncertainty!

There are several different models for document search; the one implemented in PostgreSQL is called a Boolean model. Note that modern Internet search engines use more complex models.

In a Boolean search model, a document is considered a list of *terms*. Each term usually corresponds to a word in the document transformed using certain linguistic tools. This conversion is needed to improve the quality of the search. For example, we expect that the words “word” and “words” should map to the same term. The transformations are not trivial: “a leaf” and “leaves” are the forms of the same word; however, “to leave” is different. And it’s not just morphological transformations: meaning depends on context. For example, “host” and “computer” have the same meaning in a document describing network protocols, but these words are different in a document related to organizing a conference.

In PostgreSQL, the linguistic rules defining the transformations are encapsulated in a *configuration*. Several predefined configurations for different languages are available, and additional configurations may be defined. One of the predefined configurations does not depend on any language.

The output of this linguistic processing is represented as a value of type `ts_vector`. The values of `ts_vector` are lists of terms and not related to any language or even text. A `ts_vector` can be built from any list of values.

Why is text search called *full text search*? In the dark ages (the 1970s), when the capacity of hard drives was small, lists of terms were built from titles or abstracts. So full text means that all the words in the document are considered as a source of terms.

Similarly, a query for document search is represented as a value of type `ts_query`. These values are constructed from a textual representation of a query that can contain words and the logical operators AND, OR, and NOT. A simple query consists of words only (with an implicit AND between them).

A term (word) in a query yields true if it is present in the document and yields false otherwise. The Boolean values for the terms are combined into a single Boolean value using operators contained in the query. For simple queries that only contain terms, a document matches if all terms in the query are present in the `ts_vector` corresponding to the document.

The match operator `@@` returns true if a document satisfies the query and false otherwise. It can be used in the WHERE clause of a SELECT statement or anywhere else where a Boolean expression is expected.

A Boolean search produces definite results: a document either matches a query or does not match. Is it still uncertain? Yes, it is. Some information is lost when a document is converted to `ts_vector`, and some information is lost when a query is converted into `ts_query` as well.

The text search features of PostgreSQL can work without any indexes. PostgreSQL also provides special types of indexes that can speed up text search. We discuss these index types later in this chapter.

Multidimensional and Spatial Search

Filtering conditions considered in previous chapters used scalar attributes. Some kinds of indexes, for example, compound indexes, may contain several attributes, but the ordering of the attributes is essential: the index is useless if the value of the first attribute is not specified.

A fixed ordering of attributes is not desirable for some applications and data types. These applications require that multiple attributes are treated symmetrically, without any preference to any attribute. Typical examples are objects on a plane or three-dimensional space with coordinates as attributes. These kinds of data are collectively called spatial.

More importantly, spatial data often requires different types of queries. The most common queries are

- *Range queries* – Find all objects located at a certain distance or closer to the specified point in space.
- *Nearest-neighbor queries* – Find the k objects closest to the specified point.

These queries cannot be supported with one-dimensional indexes even if multiple indexes are used.

Of course, this kind of search is not limited to space only. The coordinates might have timestamp values or even values in a discrete domain.

PostgreSQL provides index types suitable for spatial data. These indexes are briefly discussed in the next section.

Generalized Index Types in PostgreSQL

The `CREATE INDEX` operator includes an optional index type specified on index creation. So far, this hasn't been demonstrated in prior examples, because all of the previously created indexes were B-tree indexes, and B-tree is the default value for the index type. The other possible values are hash, GIST, spgist, GIN, BRIN, and user-defined type (if installed), at the time of writing.

This section discusses some of these types in more detail.

GIST Indexes

Some applications use multi-attribute objects like points with coordinates on a surface or a plane. Previously, we discussed compound indexes that include multiple attributes. However, compound indexes are not symmetric: the attributes are ordered. In contrast, the coordinates of a point should be treated symmetrically.

GIST is a family of index structures, each of which supports a certain data type and can be configured to implement several different tree-based index structures. Specifically, it implements index structure for spatial data known as an R-tree. Support for R-tree and a few other indexes is included in the PostgreSQL distribution; more types may be installed as extensions.

An R-tree implemented as the GIST index type does exactly what is needed here: it indexes spatial objects (such as points and rectangles). Objects that have more complex shape are represented in the index with rectangular bounding boxes. Search conditions are expressed with a rectangle. All objects that intersect with the search rectangle are returned as the result of the search.

Of course, the attributes included in a GIST index are not necessarily coordinates. Such attributes may represent time or other ordered data types.

Indexes for Full Text Search

PostgreSQL provides two kinds of indexes that can support text search. First, consider GIN indexes, where GIN stands for Generalized Inverted.

For the purpose of indexing, a document is considered as a list of terms (or tokens), for example, as a value of the `ts_vector` data type discussed previously.

For each term contained in at least one document, an inverted index contains a list of documents containing the term. Thus, the overall structure is symmetrical: a document has a list of terms, and a term has a list of documents. This symmetry explains why the index type is called inverted.

Inverted indexes can efficiently support text search. For example, to find documents containing all terms specified in a query, PostgreSQL scans all lists of documents for these terms and leaves only the documents that appear in the lists for all terms in the query. The lists are ordered, so a single pass over the lists is sufficient to produce the result set.

A GIN index can be created as a functional index with an expression converting the document being indexed into a `ts_vector`, or values of `ts_vector` can be stored as a separate column. The advantage of the former approach is that it uses less space, while the advantage of the latter is that the index does not depend on the configuration (as the configuration is needed only to compute the value of `ts_vector`). If the values of `ts_vector` are stored, an index can refer to documents written in different natural languages and converted into `ts_vector` with different configurations. Generally, pre-computed `ts_vectors` will perform better, but this can depend as always on the function used to generate the vectors and system configuration.

The structure of GIN is not derived from or related to a natural language; as noted earlier, it treats documents as lists. Thus, it can work with data other than documents—in fact, any attribute type containing multiple values, such as arrays. The GIN index will find all rows with multivalued attributes containing all values specified in a query, just as it finds all documents containing specified terms.

Documents (values of `ts_vector` type) can also be indexed with GIST. To build such an index, the values of `ts_vector` are converted into bitmaps of fixed length. To construct the bitmap, each term that appears in any of the documents being indexed is hashed to obtain a number that represents its position in the bitmap. Each term is a bitmap of the same length, with a single bit equal to 1 (representing the term) and all other bits set to zero. A bitmap for a document is a bitwise logical OR of all the bitmaps that represent terms in the document. Thus, the document bitmap has a bit equal to 1 in all positions corresponding to terms appearing in that document. A query bitmap is constructed similarly.

A search on this index is based on the following fact: a document satisfies the query if its bitmap contains 1 in every position that the query bitmap has a 1.

Different terms can be hashed into the same position. Therefore, a GIST index can return documents that are not relevant for the query; and usually, a recheck of `ts_vector` values is needed, but PostgreSQL can recognize this automatically.

The number of false matches grows as the number of different terms increases. Therefore, the GIST index is efficient for collections of documents where the total number of different terms is small. This is uncommon with texts in natural languages, so GIN indexes are usually more efficient in this case. GIST indexes for textual search are still useful in special cases.

Indexing Very Large Tables

Any index occupies some space, and indexes on large tables can be very large. Is it possible to reduce the size of an index?

Database textbooks distinguish between dense and sparse indexes. All the indexes covered so far are dense; that is, they contain all values of the indexed column (or columns). A sparse index contains only a fraction of all values but still reduces the number of reads needed to find any value of the indexed attribute. This differs from conditional indexes that do not speed up search of values not included in the index.

Some database systems always store table rows in the order of the (surrogate) primary key. The index on a primary key can then contain only one value per table block, and hence it can be a sparse index. More advanced database systems support ordering tables on values of other attributes, which do not necessarily need to be unique. This organization of tables also provides for sparse indexes. Sometimes these indexes are called cluster indexes as the rows with the same value of the indexed column are placed close to each other.

PostgreSQL does not provide any means for explicitly controlling row ordering. However, in many cases, rows are ordered naturally. For example, rows registering certain kinds of events will be, most likely, appended to the table and will be naturally ordered on the arrival timestamp, making sparse indexing possible.

A generalization of sparse indexes implemented in PostgreSQL is called BRIN (for Block Range Index). A table for which a BRIN index is created is considered a sequence of block ranges, where each range consists of a fixed number of adjacent blocks. For each range, a BRIN index entry contains a summary of column values contained in the block range. For example, a summary may contain the minimum and maximum values of the timestamp column in an event log table.

To find any value of the indexed attribute, it is sufficient to find an appropriate block range (using the index) and then scan all blocks in the range.

The structure of the summarization method depends on the type of the column being indexed. For intervals, a summary may be an interval containing all intervals contained in the block range. For spatial data, a summary can be a bounding box containing all boxes in the block range.

If the column values are not ordered or rows are not ordered in the table, a scan of a BRIN index will return multiple block ranges to be scanned.

The summarization is expensive. Therefore, PostgreSQL provides multiple choices for BRIN index maintenance: a BRIN index can be updated automatically with triggers; alternatively, delayed summarization can be done automatically together with vacuum or started manually.

Indexing JSON and JSONB

Sometimes developers looking for flexibility convert table rows into text or semistructured format (JSON or XML) and then use text search instead of more specific indexes. This approach definitely works better than external indexing tools, but is significantly slower than specific indexes.

Returning to the question we posed at the end of Chapter 14, why go to the trouble of building functions that transform search results into JSON when we can simply store the JSON type directly in the database?

Let's see how this approach would work in practice. To do so, let's build a table that stores bookings as JSON objects. The first problem we encounter is that we might need different JSON structures for different application endpoints (we already built several different record types in Chapters 12–14). But let's assume we can consolidate different requirements and store the data in a way that would satisfy most use cases. We can use the code in Listing 15-1.

Listing 15-1. Building a table with JSONB column type

```
--create simplifies booking leg type
CREATE TYPE booking_leg_record_2 AS
(booking_leg_id integer,
  leg_num integer,
  booking_id integer,
  flight flight_record);
--create simplified booking type
```

```

CREATE TYPE booking_record_2 AS
(
    booking_id integer,
    booking_ref text,
    booking_name text,
    email text,
    account_id integer,
    booking_legs booking_leg_record_2[],
    passengers passenger_record[]
);
---create table
CREATE TABLE booking_jsonb AS
SELECT  b.booking_id,
to_jsonb ( row (
    b.booking_id, b.booking_ref, b.booking_name, b.email, b.account_id,
ls.legs,
ps.passengers
) :: booking_record_2 ) as cplx_booking
FROM booking b
JOIN
    (SELECT booking_id,    array_agg(row (
booking_leg_id, leg_num, booking_id,
    row(f.flight_id, flight_no, departure_airport,
    dep.airport_name,
arrival_airport,
arv.airport_name,
    scheduled_departure, scheduled_arrival
)::flight_record
)::booking_leg_record_2)  legs
FROM  booking_leg l
JOIN flight f   ON  f.flight_id = l.flight_id
JOIN  airport dep ON dep.airport_code = f.departure_airport
JOIN  airport arv ON arv.airport_code = f.arrival_airport
GROUP BY booking_id)  ls
ON b.booking_id = ls.booking_id
JOIN
( SELECT    booking_id,

```

```

array_agg(
  row(passenger_id, booking_id, passenger_no, last_name, first_name)::
passenger_record) as passengers
FROM passenger
GROUP by booking_id) ps
ON ls.booking_id = ps.booking_id
) ;

```

Note that we create the table with a column type of JSONB (JSON Binary), not JSON. The only difference between these types is that JSONB stores a binary representation of the JSON data, rather than a string. For the JSON type, the only indexes you can build are B-tree indexes on specific tags, and then you need to specify a full path, including the indexes in the arrays, which would make it impossible, for example, to index “any” booking leg.

If we want to build highly performant indexes on JSON columns, we need to use the JSONB type.

Building this table will take a while. And it will take a while to build a GIN index:

```
CREATE INDEX idxgin ON booking_jsonb USING GIN (cplx_booking);
```

However, after this index is created, it feels like all our problems are solved. Now we can retrieve all the data we need without any joins and any complex structure builds, using simple queries like the one shown in Listing 15-2.

Listing 15-2. Search using a GIN index on a JSONB column

```

SELECT *
FROM booking_jsonb
WHERE
cplx_booking @@ '$.**.departure_airport_code == "ORD" && $.**.arrival_
airport_code == "JFK"'

```

The execution plan in Figure 15-1 proves that the GIN index is used.

QUERY PLAN	
	text
1	Bitmap Heap Scan on booking_jsonb (cost=199.73..21438.53 rows=5643 width=1205)
2	Recheck Cond: (cplx_booking @@ ('\$.**.departure_airport_code' == 'ORD' && \$.**.arrival_airport_code' == 'JFK'))::jsonpath)
3	-> Bitmap Index Scan on idxgin (cost=0.00..198.32 rows=5643 width=0)
4	Index Cond: (cplx_booking @@ ('\$.**.departure_airport_code' == 'ORD' && \$.**.arrival_airport_code' == 'JFK'))::jsonpath)

Figure 15-1. Execution plan with a GIN index

There are several issues that make this approach less appealing than it looks at first glance. First, this search is still slower than the search that uses B-tree indexes. The search functions generated using techniques described in Chapter 14 produce the same result but two to five times faster.

Second, GIN indexes do not support searches on date-time attributes or searches using the `like` operator or searches on any transformed attribute values, like `lower()`. You can specify several complex search conditions with `json_path` expressions and JSONB operators and functions, including regular expressions, in the `WHERE` clause, but these will be checked with a heap scan. A good idea is to combine such conditions with others that are supported with indexes.

In fact, we saw a production system that was built this way: data from multiple tables/schemas was used to create “search documents” of type JSON, and then `ts_vector` columns were added and indexed.

However, there is a third problem with this approach. As already stated, one JSON structure would support only one hierarchy. If we built a `booking_jsonb` column as described earlier, we could relatively easily update a flight in the booking leg, but we couldn’t update the actual departure time or flight status.

This means that the `booking_jsonb` table will have to be rebuilt periodically in order to remain useful. Indeed, the production system mentioned earlier had a complex sequence of triggers that rebuilt all potentially affected JSON data. In cases with a relatively low expected number of updates, this restriction might not be critical, but that is not the case with delayed flights and changing flight schedules.

Summary

PostgreSQL has a multitude of different indexes. This book covers many of them, but not all of them; new index types appear with nearly every new version. It would be unsurprising if new indexes are released between the writing of this chapter and this book’s release.

Both Chapter 5 and this chapter provide a number of examples of how to choose the right indexes to support different searches. Choosing indexes that are best suited for your system, for specific searches, is not a straightforward task. Do not stop at creating B-tree indexes for individual columns. Do you need compound indexes? Functional indexes? Will a GIST index help solve your problem? How critical is response time for this particular query? How much does this particular index impact updates? Only you can answer these questions.

CHAPTER 16

Ultimate Optimization Algorithm

The preceding chapters covered a lot of optimization techniques: not only different ways to optimize SQL statements but also how database design affects performance, the importance of working together with application developers, the use of functions, and many other aspects of database performance.

Still, the question posed in the Introduction remains: Where to start, when you have a real-world problem, when your users see an hourglass and you have no idea why? A related but more challenging task is to figure out what to do from the start. You do not have a problem yet. You have a task, possibly a draft of a query, or maybe you are lucky enough to have detailed requirements. How do you make sure you are doing it right?

In this chapter, we will present a step-by-step guide that will help you write queries right right away and, when you have the option, choose the database design that is right for you.

Major Steps

Figure 16-1 presents a flowchart that we suggest you can use to identify the best strategy for your query in question. In the subsequent sections, we will discuss each step in more detail.

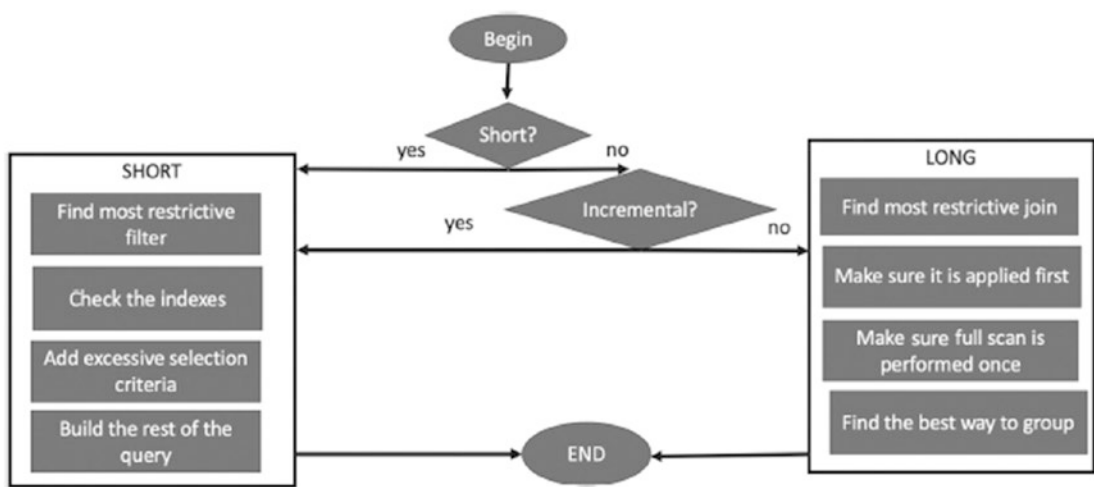


Figure 16-1. Steps of the Ultimate Optimization Algorithm

Step-by-Step Guide

Step 1: Short or Long?

The first step is to determine whether the query in question is short or long. As discussed in Chapters 5 and 6, looking at the query itself won’t necessarily help you find the answer. Step 1 is a great time to recall that query optimization starts from gathering requirements, and it’s important to work together with business owners and/or business analysts.

Check whether the business is interested in the most recent data or they need to follow historical trends and so on. The business might say that they need to see all canceled flights, but it would be a good idea to ask whether they want to see all canceled flights from the beginning of time or within the past 24 hours.

If you determine that the query in question is short, go to step 2; otherwise go, to step 3.

Step 2: Short

So your query is a short query. Which steps do you need to follow to make sure that not only is it written in the best possible way but also that query performance will be stable even when data volumes grow?

Step 2.1: The Most Restrictive Criteria

Find the most restrictive criteria for your query. Remember that often you can't tell which criteria this will be by just looking at the query. Query the tables to find the number of distinct values of attributes. Be aware of the value distribution (i.e., find out which values are the least frequent). When the most restrictive criteria are identified, proceed to the next step.

Step 2.2: Check the Indexes

In this step, you need to check whether there are indexes that support the search on the most restrictive condition. This includes the following:

- Check whether all search attributes for the most restrictive condition are indexed. If the index(es) is missing, request or create one.
- If more than one field is involved, check whether a compound index would perform better and whether the performance gains are enough to justify the creation of an additional index.
- Check whether you can use an index-only scan using either a compound or covering index.

Step 2.3: Add an Excessive Selection Criterion, If Applicable

If the most restrictive condition is based on a combination of attributes from different tables and thereby can't be indexed, consider adding an excessive selection criterion.

Step 2.4: Building (or Rebuilding) the Query

Start writing the query by applying the most restrictive criteria; this may mean starting from a select from a single table or a join that incorporates the most restrictive criteria.

Do not omit this step. Often, when database developers know the relationships between objects, they tend to write all the joins before applying filtering. While we are aware that this is an often-recommended approach, we believe that for complex queries with multiple joins, it might complicate development. We suggest starting from a SELECT that you know is executed efficiently and then adding one table at a time.

Check query performance and the execution plan each time you add a new join. Remember, optimizers tend to err in estimating the size of intermediate result sets more and more the further from the root of the execution tree they are. If the number of joins in the query is approaching ten, you may consider either using CTEs, if you are on version 12 or higher, or you may consider building dynamic SQL.

Step 3: Long

Your query is a long query. In this case, the first step would be to determine whether you can use incremental refresh. Once again, this is when you need to work together with the business owner and/or business analysts to understand better what the purpose of the query is. Often, requirements are formulated without considering data dynamics. When the results of a query are stored in a table and it is updated periodically, it can either be pulled fresh each time (a full refresh, pulling all data from the dawn of time to the most recently available data), or it can be pulled incrementally, bringing in only data that has changed since the last data pull. The latter is what we mean by incremental updates. In the vast majority of cases, it is possible to pull data incrementally. For example, instead of creating the `passenger_passport` materialized view as shown in Chapter 7, create it as the table `passenger_passport`, and add/update rows when new passport information is entered.

- If it is possible to use incremental updates, go to step 4.
- Otherwise, go to step 5.

Step 4: Incremental Updates

Treat the query to select recently added/updated records as a short query with time of update being the most restrictive criterion. Go to step 2 and follow the steps for optimizing short queries.

Step 5: Non-incremental Long Query

If running incremental updates is not possible, proceed with the following steps of long query optimization:

- Find the most restrictive join, semi-join, or anti-join, if applicable (refer to Chapter 6 for details), and make sure it is executed first.
- Keep adding tables to your join, one by one, and check the execution time and the execution plan each time.
- Make sure you do not scan any large tables multiple times. Plan your query to go through large tables only once, as described in Chapter 6.
- Pay attention to grouping. In the majority of cases, you need to postpone grouping to the last step, that is, you need to make sure that GROUP BY is the last statement in the execution plan. Be aware of some cases described in Chapter 6, when grouping should be performed earlier to minimize the size of intermediate datasets.

But Wait—There Is More!

Throughout this book, we’ve insisted that database optimization is not limited to optimizing individual queries; individual queries do not come from outer space. Still, the optimization algorithm described in the previous sections is a guide only to the process of optimizing an individual query or rather writing a query the right way from the start. However, we’ve covered several other techniques.

Here are other things to consider:

- *Parameters* – Most likely, the query you are optimizing is parameterized, that is, if you have a condition on `flight_id`, it won’t be `flight_id=1234` all the time, but rather could be any arbitrary `flight_id`. As we discussed in Chapter 5, depending on particular filtering values, the most restrictive criterion may differ (e.g., “Canceled” flight status will be more restrictive than most other criteria).
- *Dynamic SQL* – In the latter situation, the right approach is to use dynamic SQL, which will be also the case when the selection criteria themselves vary.

- *Functions* – As discussed in Chapter 12, functions in and of themselves do not improve performance and may significantly degrade the execution times. However, if dynamic SQL is needed, it is difficult to avoid using functions.
- *Database design changes* – While working on queries, you might feel the need to make some DDL changes, from creating new indexes to changing the table schema. You will need to work with your DBAs and system architects to determine which changes can be implemented.
- *Interaction with the application* – If your query is executed by an application, query performance might be pretty good, while the overall application performance may not be. If you and your team choose to use NORM or another similar approach, you will need to work with application developers to determine what belongs to business logic and what belongs to database logic, as described in Chapters 11 and 13.

Summary

This chapter provided a step-by-step guide to help you navigate the process of writing queries right right away. We encourage you to give it a try and follow these steps when working on your next project.

CHAPTER 17

Conclusion

Like all good things must, this book is coming to an end.

In the Introduction, we shared that we wrote this book because we felt we could not *not* write it. Countless times, we’ve met the question: “Is there a book you can recommend for getting started with PostgreSQL?”

In the preponderance of cases, those asking this question were not database development novices. PostgreSQL is not yet the database of choice for most educational institutions, and a typical developer “new to PostgreSQL” already knows how to write syntactically correct SELECT statements, but is not familiar with the particulars of PostgreSQL. This includes not just any minor language differences, but more importantly, differences in how data is stored and how queries are processed.

Yes, of course, documentation is always available, but it is not always easy to find what you need, unless you already know exactly what you are looking for. Other resources include many excellent tutorials on various subjects, as well as the blogs of leading PostgreSQL experts. Most of the time, however, they are focused on specific topics, showing off numerous great features of PostgreSQL, but not necessarily indicating where exactly they fit into the big picture.

Of course, this book does not present the full picture either—PostgreSQL has a lot to offer, and we did not attempt to provide exhaustive coverage. Rather, we approach PostgreSQL from a different perspective: we demonstrate how to make these great features work.

That being said, we hope that this book will become a go-to book for database developers who are starting to explore PostgreSQL. We also hope that those who have already been using PostgreSQL for a while will also find some useful information, perhaps some techniques that they haven’t used before.

Our goal is to give you a structure that you can use to navigate the challenges of database development and a resource you can consult in “what if” situations. Our hope is that having this book as a guide will make it easier for you to find more details in the PostgreSQL documentation.

In addition to documentation, there are many other valuable resources available, including

- [PostgreSQL Wiki](#), which contains thousands of helpful hints, usage patterns, and anti-patterns
- *The Art of PostgreSQL*, a great book to improve your SQL skills and learn about a variety of features for developing PostgreSQL-based applications
- Blogs by PostgreSQL experts (the [Planet PostgreSQL](#) aggregator is a great resource)
- [PostgreSQL IRC chat](#)

Throughout the book, we’ve tried to explain not only what to do but also why it works, because if you know the “why,” you will be able to recognize other situations in which a similar solution could work. Understanding relational theory is a key to understanding these “whys,” which is why this book began with a healthy portion of theory. For those who persevered through the theoretical chapters, your work will be rewarded. These theoretical foundations bring you one step closer to “thinking like a database,” which allows you to write your queries right right away, rather than “write first and then optimize.”

In addition, we’ve introduced the `postgres_air` schema, which is now open sourced and is available at https://github.com/hettie-d/postgres_air. We hope this realistic dataset will be helpful as a training, experimentation, and demonstration tool, as well as an educational resource.

In the Introduction, our target audience was described as IT professionals working in PostgreSQL who want to develop performant and scalable applications, anyone whose job title contains the words “database developer” or “database administrator,” or anyone who is a backend developer charged with programming database calls. We hope that one of the takeaways from this book will be that collaboration between all these groups and business owners is key to developing performant applications.

Database queries do not run in a vacuum: the database is a service. Database work is invisible if everything works well, and it is extremely visible if something goes wrong. With that in mind, we hope that your work will remain mostly invisible!

And now have fun with PostgreSQL!

Index

A

- Agile teams, 304
- Algorithms, 23, 25–45, 55
- ANALYZE command, 59, 149
- AND operation, 30
- Anti-join
 - execution plan, 117
 - with execution plan, 117
 - JOIN operator, 118–120
 - NOT EXISTS keyword, 117
 - NOT IN, 117
 - without execution plan, 117
- Application developers, 258, 292–294, 315, 327, 332
- Application development patterns, 11, 220–222
- Associativity, 21
- ATTACH and DETACH partitions, 173
- Autovacuum, 173
- AUTOVACUUM system parameters, 181

B

- Bitmap index, 30, 52, 92
- Boolean search model, 318
- B-tree index, 36–39, 70, 75, 76, 314, 320
- Built-in functions, 232
- Business logic, 162, 221, 258, 259

C

- Cartesian product, 20
- CASCADE keyword, 168

- coalesce() function, 74, 94
- Code maintenance, 244
- Column transformation
 - B-tree indexes, 70
 - coalesce() function, 74
 - different search conditions, 71
 - execution plan, 73
 - execution plan for query, 71
 - functional index, 72
 - last_name column, 72
 - search criteria, 70
 - search criterion, 70
 - SELECT statement, 72
 - sequential scan, 72
 - transformed value, 70
 - update_ts::date=CURRENT_DATE, 73
- Combining relations
 - hash-based algorithms, 42, 43
 - nested loop, 40–42
 - sort-merge algorithms, 43, 44
- COMMIT/ROLLBACK command, 262
- Common table expressions (CTEs)
 - definition, 150
 - execution plan, 151, 153, 155
 - forced materialization, 154
- INSERT/DELETE/UPDATE
 - statements, 151
- MATERIALIZED and NOT
 - MATERIALIZED keywords, 152
- purpose, 151
- sample query, 150
- SELECT statements, 151

INDEX

- Commutativity, 21
- Comparison operators, 2
- Complex query, 3, 22, 54, 84, 226
- Composite types
 - array_agg(), 254
 - boarding_pass_flight function, 253
 - booking_leg_record type, 251
 - complex object, 251
 - function parameter, 253
 - internal function, 253
 - nested structure, 250
 - PostgreSQL, 254
 - record type definitions, 251
- Compound indexes, 78, 80
 - searches by arrival_airport/scheduled_departure, 79
 - covering indexes, 82, 83
 - for data retrieval, 81, 82
 - execution plan, 79
 - for first two columns, 80
 - lower selectivity, 80, 81
 - plan with index-only scan, 81
- Concurrency control
 - high-performance database, 178
 - lock waiting, 177
 - MVCC, 178
 - SI protocol, 179
 - transaction, 179
 - two-phase locking protocol, 179
 - VACUUM operation, 179
 - WAL, 177
- Connections, 207, 272
- Correctness, 54, 177, 193–195
- Cost-based optimization
 - algorithm, 47, 55
- Cost model, 26, 31
- Covering indexes, 82, 83
- CREATE FUNCTION command, 232, 262

- CREATE FUNCTION operator,
 - 242, 256, 257
- CREATE FUNCTION statement, 240, 241
- CREATE PROCEDURE command, 262
- CREATE TABLE statement, 182
- Custom conversion function, 194, 238
- custom_field table, 140, 167, 190, 194

D

- Data access algorithms
 - compare, 32
 - cost and query, 32–34
 - full scan, 28, 29
 - index-based table access, 29–31
 - index-only scan, 31
 - storage structures, 27, 28
- Database design, 7, 9, 185, 189, 327, 332
- Database developers, 8, 22, 34, 195, 227, 277, 292, 304, 315, 333
- Database DML functions, 221
- Database management systems, 17
- Database objects, 27, 29, 102, 147, 168, 257, 293, 305
- Database optimization, 11, 87, 217, 331
- Database optimizer, 2
- Database queries, 217, 334
- Database systems, 162, 322
- Data definition language (DDL), 175
- Data manipulation functions, 304
- Data manipulation language (DML)
 - commands, 175
 - concurrency control impact, 177–180
 - execution parts, 175
 - function, 257
 - and indexes, 180
 - low-level input/output, 176, 177
 - OLTP systems, 176

- optimization, 176
- SELECT statement, 176
- and vacuum, 181, 182
- Date/time conversion functions, 194
- Declarative language, 2
- Declarative queries, 22
- Declarative query languages, 219
- DECLARE clause, 235
- Denormalized data structure, 196
- Distributivity, 21
- Document, 317, 321
- Dollar quoting, 234, 235
- DROP command, 168, 173
- Dynamic functions, 286
- Dynamic SQL
 - definition, 265
 - execution plan, 266
 - booking_ids, 289
 - connection pool, 272
 - create function, parameters, 267, 268
 - function behavior, 272
 - function calls, 271, 272
 - hash joins, 268–270
 - nested loop, 270, 271
 - planning time, 273
 - quote_literal() function, 273
 - return type, 267
 - select_booking_leg_country_
 - dynamic function, 272, 273
- flexibility
 - arbitrary list, 277
 - booking, 278
 - full search criteria, 285
 - function executions, 283
 - function source code, 283–285
 - parameters, 285
 - performance, 286

- query, 285
- search fields, 277
- search, parameters, 278–280, 282, 283
- sequence of calls/execution times, 279
- optimizer, 286, See Optimizer
- vs. PostgreSQL, 265, 266
- SQL injection, 266
- SQL statement, 290

E

- End user interface, 221
- Entity-attribute-value (EAV) model, 138, 190
- Equivalence rules
 - associativity, 21
 - commutativity, 21
 - distributivity, 21
- Equivalent expressions, 22
- Exception processing section, 238
- Excessive selection criteria, 83, 87
 - execution plan, 88
 - filtering conditions, 83, 84
 - PostgreSQL, 83
 - search space, 87
 - short query with hard-to-optimize filtering, 85
 - suboptimal execution plan, 86
 - tables with query rewrite, 84, 85
- Execution plan
 - arrows, 50
 - cost, 56, 57
 - graphical representation, 51
 - index-based, 58
 - logical plan, 48
 - nodes, 50

INDEX

Execution plan (*cont.*)

- optimization, 54
- optimizer, 59, 60
- reading, 48, 50
- sorting, 53
- sub-plan, 55

Execution time, 214

EXPLAIN command, 49, 52, 53, 121, 209,
242, 268, 289

External metrics, 25

F

fillfactor parameter, 182

Filter operation, 18, 52

First commit wins, 179

First update wins, 179

Flexibility *vs.* efficiency and
correctness, 193–195

Foreign data wrappers (FDWs), 193,
289, 290

Foreign key constraint, 67, 68

Foreign keys, 67, 68

Frequent updates, 182

Full text search, 318

Function creation

- dollar quoting, 234, 235
- internal functions, 232
- procedural language, 233, 234
- user-defined, 232, 233

Function execution, 238, 239

Function execution internals, 240–242

Function outputs, 235

Function overloading, 235–237

Function parameters, 235

Functions

- composite types, 247–250
- data manipulation, 255–257

modern programming languages, 231

and performance, 243–245

portability impact, 231

and security, 257, 258

type dependencies, 255

user-defined data types, 246

G

geqo_threshold configuration, 56

get_flight_by_departure_airport(), 222

Graphic tools, 50

GROUP BY, 124, 131

H

Hash index, 39

Hash join algorithm, 42, 43, 108, 126

Heap, 28, 181, 213, 326

Heuristics, 47, 48

Hierarchical structures, 192

Hybrid hash join, 43

I

Identifiers, 16, 197, 198

Impedance mismatch, 219, 227, 291

Imperative language, 2, 16, 232, 243

Imperatively constructed query, 4

Imperative programming languages, 219

Indexes, 29

access to rows, 36

avoid index usage, 94

B-tree, 36–39

and column transformations, 70–74

create new index, 101, 102

definition, 34

hash, 39

- and join order, 90–92
- and “like” Operator, 75, 76
- partial indexes, 89, 90
- PostgreSQL modify execution plans, 97, 98
- prevent PostgreSQL, 94–97
- R-tree, 39
- and short query scalability, 103
- structures, 35
- Indexing**
 - booking_jsonb table, 326
 - database systems, 322
 - GIN index, 326
 - issues, 326
 - JSON, 323, 326
 - JSONB
 - execution plan, 325
 - GIN index, 325
 - vs.* JSON, 325
 - table creation, 323, 324
 - large tables, 322, 323
 - production system, 326
 - sparse index, 322
 - summarization method, 323
- Index-only-scan data retrieval method, 81
- Index selectivity, 64–66, 68
- Index types
 - CREATE INDEX operator, 320
 - full text search, 320, 321
 - GIN, 321
 - GIST, 320, 322
- Interfaces, 224, 225
- Internal functions, 232
- Internal metrics, 26
- Inverted indexes, 321
- I/O operations, 26
- J**
 - Join condition, 41, 151, 214
 - Join order
 - anti-join, 116, 117
 - common table expressions, 122
 - execution plan, two semi-joins, 114
 - execution plan with disabled cost-based optimization, 122
 - execution plan with two semi-joins, different selectivity, 115
 - EXPLAIN command, 121
 - force, 121
 - forces sequential scan, 116
 - join_collapse_limit, 120, 121
 - JOIN operator
 - anti-join, 118–120
 - semi-join, 118–120
 - partially disable cost-based optimization, 121
 - semi-join, 110, 111
 - two semi-joins, one long query, 113
- JSON column, 194, 195, 325
- JSON/JSONB data types, 229
- K**
 - Key-value model, 191, 192
- L**
 - Like operator, 71, 75, 76, 326
 - Linguistic processing, 318
 - Linguistic rules, 318
 - Lock waiting, 177
 - Logical operations, 21
 - Logical plan, 49

INDEX

Long queries

- complex condition rewrite with OR,
UNION ALL, 137

- EAV table, 143

- EXCEPT, 134

- EXCEPT, execution plan, 135

- full scans, 107

- grouping

 - condition, 129

 - execution plan, 129–133

 - filter first, 123–130

 - group first, 131–133

 - group last, 123–130

 - select last, 131–133

 - to subquery, 143, 145

- hash joins, 108

- INTERSECT, 135

- INTERSECT, execution plan, 136

- join order (*see* Join order)

- with large result set, 106

- multiple scans

 - execution plan, 140

 - large table, 139

 - postgres_air schema, 138

- multiple table scans, 138

- OLAP systems, 120

- with one-row result set, 106

- one table scan, 141, 142

- query with complex selection criteria
with OR, 136

- SET operations, 134–138

- Long query, 62, 63

M

- Maintenance operations, 208

- Mapping complex objects, 229

- Materialized views

 - conditions, 164–167

 - creation, 162, 163

 - definition, 162

 - dependencies, 167, 168

 - optimization, 167

 - refreshing, 164

 - storing results, 162

- Medical record numbers (MRNs), 197

- Memory allocation parameters, 206

- Missing index, 213

- Multidimensional search, 319

- Multiple-hierarchy approach, 193

- Multiple indexes, 30, 77, 78

- Multiple marketing research studies, 218

- Multi-version concurrency control
(MVCC), 178, 179

N

- Nested loop algorithm, 40, 42, 68, 108

- Nested loop join algorithm, 41, 68,
114, 222

- Nonoptimal database design, 189

- Non-relational database system, 190

- Non-relational solutions, 195

- No object-relational mismatch (NORM)

 - application/database, 300, 301

 - application developers, 292, 315

 - application perspective

 - JSON, 304

 - manifest, transfer object, 303

 - Python classes, 301, 302

 - booking, 295

 - contract, 296, 300

 - database perspective

 - database code, 306, 307

 - data modification, 311, 313

 - filtering conditions, 310

- JSON, 305, 306
 - NORM-GEN, 304
 - search conditions, 310
 - SELECT clause, 308, 309
 - database tables, 296
 - data exchange, 315
 - ERD, 294, 295
 - flight data, 296
 - GitHub, 291
 - JSON, 296–299, 314
 - object definitions, 296
 - vs.* ORM, 292–294
 - performance, 314
 - transfer object, 300
 - transport object hierarchy, 295, 296
 - Normalization
 - dead theory, 195
 - ER model, 196
 - improving performance, 196
 - postal address, 196
 - purpose, 196
 - USPS kiosk, 196
 - num_passengers(int) function, 243
- O**
- Object-oriented programming (OOP), 147, 277
 - Object-relational impedance mismatch (ORIM), 227, 291
 - Object-relational mapper (ORM)
 - abstraction problem solving, 226
 - definition, 225
 - impedance mismatch
 - problem, 225
 - implementation details, 226
 - object IDs, 226
 - process, 225, 226
 - running arbitrary database queries, 226
 - OLTP queries, 64
 - OLTP systems, functions
 - dynamic SQL execution ability, 261
 - no explicit dependencies, 261
 - parameterizing, 260
 - One-table design, 186–188
 - One-table solution, 187
 - Online Analytical Processing (OLAP), 8
 - age_category function, 274, 277
 - age category assignment, 275
 - age_category_dyn function, 276, 277
 - age_category function, 275
 - analytical queries, 275
 - code as text, 275, 276
 - code factoring, 277
 - create function, 277
 - execution time, 275
 - statistical reports, 274
 - Online booking system, 227
 - Online Transaction Processing (OLTP), 8, 102, 189
 - Optimization, 1
 - Optimization algorithm
 - incremental updates, 330
 - long query, 328, 330
 - non-incremental long query, 330, 331
 - short query, 328–330
 - steps, 327, 328
 - techniques, 331, 332
 - Optimization fence, 151
 - Optimizer
 - booking/departure airport, 286–288
 - booking_ids, 287, 288
 - execution plan, 287
 - execution time, 287
 - suboptimal plan, 287
 - Order of joins, 91, 98

P

- Parallel execution, 174
- Parallelism, 173, 174
- Parallel processing, 174
- Parameter tuning
 - default memory allocation, 209
 - execution plan, 210, 211
 - EXPLAIN ANALYZE, 211
 - EXPLAIN command, 209
 - limitations, 216
 - optimization query, 209
 - shared_buffers, 211–213
 - total execution time, 211
 - work_mem, 213
- Partial indexes, 89, 90, 194
- Partitioned table, 168
- Partitioning
 - description, 168
 - improving query
 - performance, 170–172
 - large data distribution, 173
 - PostgreSQL, 168
 - primary reason, 173
- Pattern search index, 76
- Performance metrics, 219
- Pessimization, 129
- Physical operations, 23, 25, 26
- PL/pgSQL functions, 255
- PL/pgSQL interpreter, 241
- postgres_air database, 95, 208, 227
- PostgreSQL, 12, 13
 - database developers, 333
 - database development, 333
 - documentation, 333, 334
 - execution plans, 265
 - functions, 268
 - IT professionals, 334

- optimization step, 266
 - postgres_air schema, 334
 - relational theory, 334
 - SELECT statements, 333
- PostgreSQL configuration parameters
 - connections and sessions, 206–208
 - database developer, 205
 - database performance, 205
 - memory allocation, 206
 - tuning, 208–213
- PostgreSQL expression, 22
- PostgreSQL features, 229
- PostgreSQL optimizer, 224
- Predefined configurations, 318
- Prepared statement, 242, 244
- Procedural languages, 233, 234
- Product operation, 20
- Project operation, 19

Q

- Query
 - casting date, 2
 - compiler, 16
 - complex, 54
 - declarative, 5
 - execution time, 12
 - operators, 2
 - simple, 54
 - total passengers, 4
- Query compiler, 16
- Query selectivity, 64

R

- random_page_cost parameter, 216
- Range partitioning, 168
- Redundancy, 34

Referential integrity, 182, 183
 REFRESH command, 164
 Relational algebra, 17
 Relational databases, 189
 Relational models
 database types, 190
 definition, 190
 EAV, 190
 FDWs, 193
 hierarchical structures, 192
 key-value model, 191, 192
 Relational operation, 17, 21
 Response time, 217, 218
 Responsibly, 193
 RETURNS clause, 234
 RETURNS VOID function, 235
 R-tree index, 39

S

Scalar functions, 250
 SECURITY DEFINER function, 258
 select rules, 162
 select list, 243
 select operator, 237
 Selectivity, 27
 SELECT list, 245
 SELECT statement, 147
 Semi-join, 116, 117
 conditions, 110
 definition, 109
 execution plan, 110, 111
 EXISTS keyword, 110
 IN keyword, 110
 JOIN operator, 118–120
 join order, 111, 112
 Sequential scan, 112
 Sessions, 208
 Shopping list problem, 222–224
 Short queries, 8, 62, 63, 145
 Simple query, 54
 Single-table design, 10
 Small percentage, 63
 SMART goals, 7
 Snapshot isolation (SI), 179
 Software engineering architecture
 pattern, 220
 Sorting, 53
 Sort-merge algorithm, 43, 44
 Spatial search, 319
 SQL injection, 266
 SQL operators, 233
 SQL query
 compiling, 15
 optimization, 16
 SQL statements, 249
 Stored procedures
 exception processing, 263, 264
 vs. functions, 262
 functions with no results, 261
 transaction management, 262, 263
 Surrogate keys
 account table, 201
 advantage, 197
 aircraft table, 199
 airport table, 198
 boarding_pass table, 203
 booking_leg table, 200
 booking table, 199
 definition, 197
 extra joins, 198
 flight_id, 202
 frequent_flyer_id, 202
 internal standards, 197
 MRNs, 197
 passenger table, 200, 201

INDEX

Surrogate keys (*cont.*)

- postgres_air schema, 198
- primary key, 199
- requirement, 197
- uniqueness, 197

T

Table bloat, 181

Temporary tables

- CREATE statement, 148
- CREATE TABLE statement, 148
- creation time, 150
- inefficient usage, 149
- intermediate results, 149
- problems, 149
- storing results, 148

text_to_date function, 239

Transformation rules, 47

Triggers, 183

Trivial syntax errors, 241

Two-phase locking concurrency control
protocol, 179

Two-table design, 10, 185, 187, 188

Two-table solution, 187

U

Unique constraint, 66, 67

Unique indexes, 66–68

Mass UPDATE/DELETE, 181

User-defined conversion function, 238

User-defined data types, 255

User-defined functions, 232, 233

User-defined scalar functions, 238

V

VACUUM operation, 177, 181

Views

- column transformation, 159
- creation, 155
- definition, 155
- encapsulation, 158
- execution plan, 157, 158
- order of joins and searches, 159
- performance degradation, 160
- purpose, 162
- query, 155, 156
- query plan, 156
- SELECT statement, 155, 159, 161
- tables, 158

W, X, Y, Z

Web application, 6

World Wide Wait, 218

Write-ahead log (WAL), 176, 177