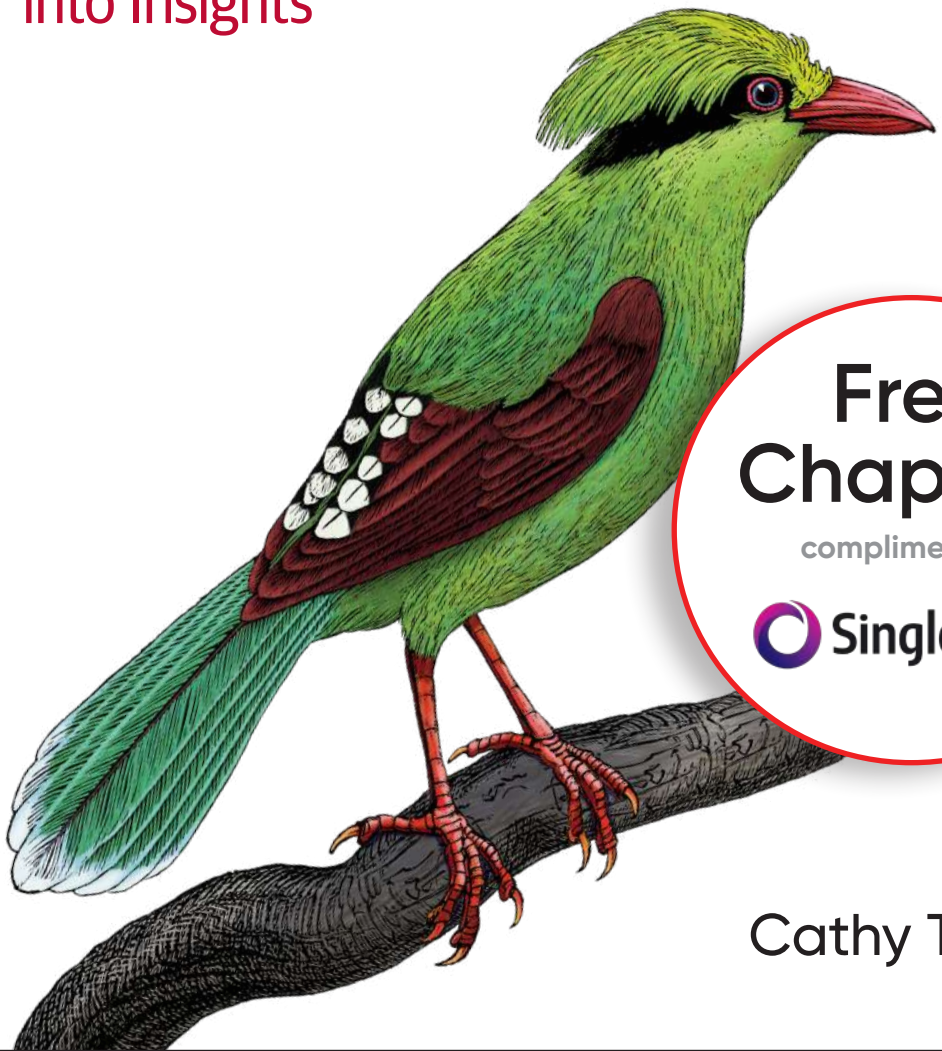
O'REILLY®

SQL for Data Analysis

Advanced Techniques for Transforming Data into Insights



Free Chapters

compliments of

SingleStore

Cathy Tanimura

SingleStore

The Single Database

For Your Data-Intensive Applications

Switch to the unified database with 10 to 100x the performance at one-third the cost of legacy infrastructures. SingleStore delivers unmatched speed, scale, and agility in one powerful, cloud-native relational database.

Pure Speed
Fast transactions, fast analytics

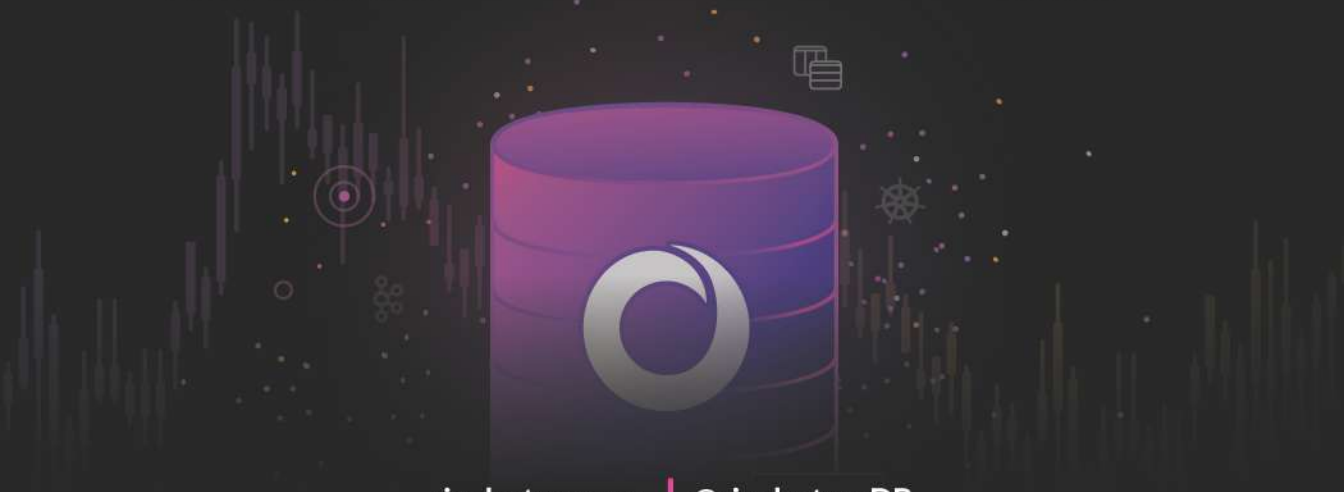
Universal Storage
Patented single table type for transactions & analytics

Fast Ingestion
Pipelines - Load data with updates

Multi-Model
Geospatial, Time-Series, Semi-Structured, & more

Run Anywhere
Your cloud or mine?

Compatibility
ANSI SQL & MySQL Ecosystem



singlestore.com | @singlestoreDB

SQL for Data Analysis

Advanced Techniques for Transforming Data into Insights

This excerpt contains Chapters 1 through 3. The complete book is available on the O'Reilly Online Learning Platform and through other retailers.

Cathy Tanimura

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

SQL for Data Analysis
by Cathy Tanimura

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Andy Kwan
Development Editors Amelia Blevins and Shira Evans
Production Editor: Kristen Brown
Copyeditor: Arthur Johnson
Proofreader: Paula L. Fleming
Indexer: Ellen Troutman-Zaig
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

September 2021: First Edition

Revision History for the First Edition
2021-09-09: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492088783* for release details.

978-1-492-08878-3

[LSI]

Table of Contents

CHAPTER 1

Analysis with SQL

If you're reading this book, you're probably interested in data analysis and in using SQL to accomplish it. You may be experienced with data analysis but new to SQL, or perhaps you're experienced with SQL but new to data analysis. Or you may be new to both topics entirely. Whatever your starting point, this chapter lays the groundwork for the topics covered in the rest of the book and makes sure we have a common vocabulary. I'll start with a discussion of what data analysis is and then move on to a discussion of SQL: what it is, why it's so popular, how it compares to other tools, and how it fits into data analysis. Then, since modern data analysis is so intertwined with the technologies that have enabled it, I'll conclude with a discussion of different types of databases that you may encounter in your work, why they're used, and what all of that means for the SQL you write.

What Is Data Analysis?

Collecting and storing data for analysis is a very human activity. Systems to track stores of grain, taxes, and the population go back thousands of years, and the roots of statistics date back hundreds of years. Related disciplines, including statistical process control, operations research, and cybernetics, exploded in the 20th century. Many different names are used to describe the discipline of data analysis, such as business intelligence (BI), analytics, data science, and decision science, and practitioners have a range of job titles. Data analysis is also done by marketers, product managers, business analysts, and a variety of other people. In this book, I'll use the terms *data analyst* and *data scientist* interchangeably to mean the person working with SQL to understand data. I will refer to the software used to build reports and dashboards as *BI tools*.

Data analysis in the contemporary sense was enabled by, and is intertwined with, the history of computing. Trends in both research and commercialization have shaped it,

and the story includes a who's who of researchers and major companies, which we'll talk about in the section on SQL. Data analysis blends the power of computing with techniques from traditional statistics. Data analysis is part data discovery, part data interpretation, and part data communication. Very often the purpose of data analysis is to improve decision making, by humans and increasingly by machines through automation.

Sound methodology is critical, but analysis is about more than just producing the right number. It's about curiosity, asking questions, and the "why" behind the numbers. It's about patterns and anomalies, discovering and interpreting clues about how businesses and humans behave. Sometimes analysis is done on a data set gathered to answer a specific question, as in a scientific setting or an online experiment. Analysis is also done on data that is generated as a result of doing business, as in sales of a company's products, or that is generated for analytics purposes, such as user interaction tracking on websites and mobile apps. This data has a wide range of possible applications, from troubleshooting to planning user interface (UI) improvements, but it often arrives in a format and volume such that the data needs processing before yielding answers. Chapter 2 will cover preparing data for analysis, and Chapter 8 will discuss some of the ethical and privacy concerns with which all data practitioners should be familiar.

It's hard to think of an industry that hasn't been touched by data analysis: manufacturing, retail, finance, health care, education, and even government have all been changed by it. Sports teams have employed data analysis since the early years of Billy Beane's term as general manager of the Oakland Athletics, made famous by Michael Lewis's book *Moneyball* (Norton). Data analysis is used in marketing, sales, logistics, product development, user experience design, support centers, human resources, and more. The combination of techniques, applications, and computing power has led to the explosion of related fields such as data engineering and data science.

Data analysis is by definition done on historical data, and it's important to remember that the past doesn't necessarily predict the future. The world is dynamic, and organizations are dynamic as well—new products and processes are introduced, competitors rise and fall, sociopolitical climates change. Criticisms are leveled against data analysis for being backward looking. Though that characterization is true, I have seen organizations gain tremendous value from analyzing historical data. Mining historical data helps us understand the characteristics and behavior of customers, suppliers, and processes. Historical data can help us develop informed estimates and predicted ranges of outcomes, which will sometimes be wrong but quite often will be right. Past data can point out gaps, weaknesses, and opportunities. It allows organizations to optimize, save money, and reduce risk and fraud. It can also help organizations find opportunity, and it can become the building blocks of new products that delight customers.

Organizations that don't do some form of data analysis are few and far between these days, but there are still some holdouts. Why do some organizations not use data analysis? One argument is the cost-to-value ratio. Collecting, processing, and analyzing data takes work and some level of financial investment. Some organizations are too new, or they're too haphazard. If there isn't a consistent process, it's hard to generate data that's consistent enough to analyze. Finally, there are ethical considerations. Collecting or storing data about certain people in certain situations may be regulated or even banned. Data about children and health-care interventions is sensitive, for example, and there are extensive regulations around its collection. Even organizations that are otherwise data driven need to take care around customer privacy and to think hard about what data should be collected, why it is needed, and how long it should be stored. Regulations such as the European Union's General Data Protection Regulation, or GDPR, and the California Consumer Privacy Act, or CCPA, have changed the way businesses think about consumer data. We'll discuss these regulations in more depth in Chapter 8. As data practitioners, we should always be thinking about the ethical implications of our work.

When working with organizations, I like to tell people that data analysis is not a project that wraps up at a fixed date—it's a way of life. Developing a data-informed mindset is a process, and reaping the rewards is a journey. Unknowns become known, difficult questions are chipped away at until there are answers, and the most critical information is embedded in dashboards that power tactical and strategic decisions. With this information, new and harder questions are asked, and then the process repeats.

Data analysis is both accessible for those looking to get started and hard to master. The technology can be learned, particularly SQL. Many problems, such as optimizing marketing spend or detecting fraud, are familiar and translate across businesses. Every organization is different and every data set has quirks, so even familiar problems can pose new challenges. Communicating results is a skill. Learning to make good recommendations and becoming a trusted partner to an organization take time. In my experience, simple analysis presented persuasively has more impact than sophisticated analysis presented poorly. Successful data analysis also requires partnership. You can have great insights, but if there is no one to execute on them, you haven't really made an impact. Even with all the technology, it's still about people, and relationships matter.

Why SQL?

This section describes what SQL is, the benefits of using it, how it compares to other languages commonly used for analysis, and finally how SQL fits into the analysis workflow.

What Is SQL?

SQL is the language used to communicate with databases. The acronym stands for Structured Query Language and is pronounced either like "sequel" or by saying each letter, as in "ess cue el." This is only the first of many controversies and inconsistencies surrounding SQL that we'll see, but most people will know what you mean regardless of how you say it. There is some debate as to whether SQL is or isn't a programming language. It isn't a general purpose language in the way that C or Python are. SQL without a database and data in tables is just a text file. SQL can't build a website, but it is powerful for working with data in databases. On a practical level, what matters most is that SQL can help you get the job of data analysis done.

IBM was the first to develop SQL databases, from the relational model invented by Edgar Codd in the 1960s. The relational model was a theoretical description for managing data using relationships. By creating the first databases, IBM helped to advance the theory, but it also had commercial considerations, as did Oracle, Microsoft, and every other company that has commercialized a database since. From the beginning, there has been tension between computer theory and commercial reality. SQL became an International Organization for Standards (ISO) standard in 1987 and an American National Standards Institute (ANSI) standard in 1986. Although all major databases start from these standards in their implementation of SQL, many have variations and functions that make life easier for the users of those databases. These come at the cost of making SQL more difficult to move between databases without some modifications.

SQL is used to access, manipulate, and retrieve data from objects in a database. Databases can have one or more *schemas*, which provide the organization and structure and contain other objects. Within a schema, the objects most commonly used in data analysis are tables, views, and functions. Tables contain fields, which hold the data. Tables may have one or more *indexes*; an index is a special kind of data structure that allows data to be retrieved more efficiently. Indexes are usually defined by a database administrator. Views are essentially stored queries that can be referenced in the same way as a table. Functions allow commonly used sets of calculations or procedures to be stored and easily referenced in queries. They are usually created by a database administrator, or DBA. Figure 1-1 gives an overview of the organization of databases.

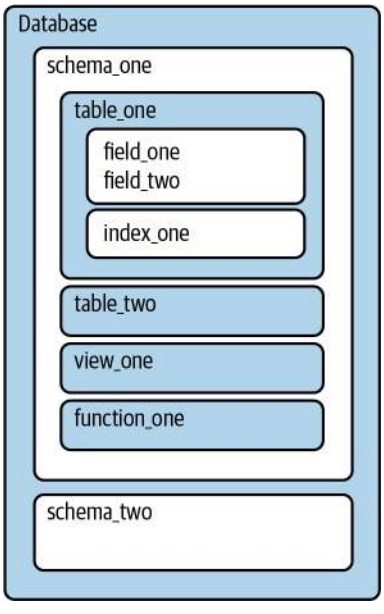


Figure 1-1. Overview of database organization and objects in a database

To communicate with databases, SQL has four sublanguages for tackling different jobs, and these are mostly standard across database types. Most people who work in data analysis don't need to recall the names of these sublanguages on a daily basis, but they might come up in conversation with database administrators or data engineers, so I'll briefly introduce them. The commands all work fluidly together, and some may coexist in the same SQL statement.

DQL, or *data query language*, is what this book is mainly about. It's used for *querying* data, which you can think of as using code to ask questions of a database. DQL commands include *SELECT*, which will be familiar to prior users of SQL, but the acronym DQL is not frequently used in my experience. SQL queries can be as short as a single line or span many tens of lines. SQL queries can access a single table (or view), can combine data from multiple tables through the use of joins, and can also query across multiple schemas in the same database. SQL queries generally cannot query across databases, but in some cases clever network settings or additional software can be used to retrieve data from multiple sources, even databases of different types. SQL queries are self-contained and, apart from tables, do not reference variables or outputs from previous steps not contained in the query, unlike scripting languages.

DDL, or *data definition language*, is used to create and modify tables, views, users, and other objects in the database. It affects the structure but not the contents. There are three common commands: *CREATE*, *ALTER*, and *DROP*. *CREATE* is used to

make new objects. *ALTER* changes the structure of an object, such as by adding a column to a table. *DROP* deletes the entire object and its structure. You might hear DBAs and data engineers talk about working with DDLs—this is really just shorthand for the files or pieces of code that do the creates, alters, or drops. An example of how DDL is used in the context of analysis is the code to create temporary tables.

DCL, or *data control language*, is used for access control. Commands include *GRANT* and *REVOKE*, which give permission and remove permission, respectively. In an analysis context, *GRANT* might be needed to allow a colleague to query a table you created. You might also encounter such a command when someone has told you a table exists in the database but you can't see it—permissions might need to be *GRANT*ed to your user.

DML, or *data manipulation language*, is used to act on the data itself. The commands are *INSERT*, *UPDATE*, and *DELETE*. *INSERT* adds new records and is essentially the "load" step in extract, transform, load (ETL). *UPDATE* changes values in a field, and *DELETE* removes rows. You will encounter these commands if you have any kind of self-managed tables—temp tables, sandbox tables—or if you find yourself in the role of both owner and analyzer of the database.

These four sublanguages are present in all major databases. In this book, I'll focus mainly on DQL. We will touch on a few DDL and DML commands in Chapter 8, and you will also see some examples in the GitHub site for the book, where they are used to create and populate the data used in examples. Thanks to this common set of commands, SQL code written for any database will look familiar to anyone used to working with SQL. However, reading SQL from another database may feel a bit like listening to someone who speaks the same language as you but comes from another part of the country or the world. The basic structure of the language is the same, but the slang is different, and some words have different meanings altogether. Variations in SQL from database to database are often termed *dialects*, and database users will reference Oracle SQL, MSSQL, or other dialects.

Still, once you know SQL, you can work with different database types as long as you pay attention to details such as the handling of nulls, dates, and timestamps; the division of integers; and case sensitivity.

This book uses PostgreSQL, or Postgres, for the examples, though I will try to point out where the code would be meaningfully different in other types of databases. You can install Postgres on a personal computer in order to follow along with the examples.

Benefits of SQL

There are many good reasons to use SQL for data analysis, from computing power to its ubiquity in data analysis tools and its flexibility.

Perhaps the best reason to use SQL is that much of the world's data is already in databases. It's likely your own organization has one or more databases. Even if data is not already in a database, loading it into one can be worthwhile in order to take advantage of the storage and computing advantages, especially when compared to alternatives such as spreadsheets. Computing power has exploded in recent years, and data warehouses and data infrastructure have evolved to take advantage of it. Some newer cloud databases allow massive amounts of data to be queried in memory, speeding things up further. The days of waiting minutes or hours for query results to return may be over, though analysts may just write more complex queries in response.

SQL is the de facto standard for interacting with databases and retrieving data from them. A wide range of popular software connects to databases with SQL, from spreadsheets to BI and visualization tools and coding languages such as Python and R (discussed in the next section). Due to the computing resources available, performing as much data manipulation and aggregation as possible in the database often has advantages downstream. We'll discuss strategies for building complex data sets for downstream tools in depth in Chapter 8.

The basic SQL building blocks can be combined in an endless number of ways. Starting with a relatively small number of building blocks—the syntax—SQL can accomplish a wide array of tasks. SQL can be developed iteratively, and it's easy to review the results as you go. It may not be a full-fledged programming language, but it can do a lot, from transforming data to performing complex calculations and answering questions.

Last, SQL is relatively easy to learn, with a finite amount of syntax. You can learn the basic keywords and structure quickly and then hone your craft over time working with varied data sets. Applications of SQL are virtually infinite, when you take into account the range of data sets in the world and the possible questions that can be asked of data. SQL is taught in many universities, and many people pick up some skills on the job. Even employees who don't already have SQL skills can be trained, and the learning curve may be easier than that for other programming languages. This makes storing data for analysis in relational databases a logical choice for organizations.

SQL Versus R or Python

While SQL is a popular language for data analysis, it isn't the only choice. R and Python are among the most popular of the other languages used for data analysis. R is a statistical and graphing language, while Python is a general-purpose programming language that has strengths in working with data. Both are open source, can be installed on a laptop, and have active communities developing packages, or extensions, that tackle various data manipulation and analysis tasks. Choosing between R and Python is beyond the scope of this book, but there are many discussions online about the relative advantages of each. Here I will consider them together as coding-language alternatives to SQL.

One major difference between SQL and other coding languages is where the code runs and, therefore, how much computing power is available. SQL always runs on a database server, taking advantage of all its computing resources. For doing analysis, R and Python are usually run locally on your machine, so computing resources are capped by whatever is available locally. There are, of course, lots of exceptions: databases can run on laptops, and R and Python can be run on servers with more resources. When you are performing anything other than the simplest analysis on large data sets, pushing work onto a database server with more resources is a good option. Since databases are usually set up to continually receive new data, SQL is also a good choice when a report or dashboard needs to update periodically.

A second difference is in how data is stored and organized. Relational databases always organize data into rows and columns within tables, so SQL assumes this structure for every query. R and Python have a wider variety of ways to store data, including variables, lists, and dictionaries, among other options. These provide more flexibility, but at the cost of a steeper learning curve. To facilitate data analysis, R has data frames, which are similar to database tables and organize data into rows and columns. The pandas package makes DataFrames available in Python. Even when other options are available, the table structure remains valuable for analysis.

Looping is another major difference between SQL and most other computer programming languages. A *loop* is an instruction or a set of instructions that repeats until a specified condition is met. SQL aggregations implicitly loop over the set of data, without any additional code. We will see later how the lack of ability to loop over fields can result in lengthy SQL statements when pivoting or unpivoting data. While deeper discussion is beyond the scope of this book, some vendors have created extensions to SQL, such as PL/SQL in Oracle and T-SQL in Microsoft SQL Server, that allow functionality such as looping.

A drawback of SQL is that your data must be in a database,[1] whereas R and Python can import data from files stored locally or can access files stored on servers or websites. This is convenient for many one-off projects. A database can be installed on a laptop, but this does add an extra layer of overhead. In the other direction, packages such as dbplyr for R and SQLAlchemy for Python allow programs written in those languages to connect to databases, execute SQL queries, and use the results in further processing steps. In this sense, R or Python can be complementary to SQL.

R and Python both have sophisticated statistical functions that are either built in or available in packages. Although SQL has, for example, functions to calculate average and standard deviation, calculations of p-values and statistical significance that are needed in experiment analysis (discussed in Chapter 7) cannot be performed with SQL alone. In addition to sophisticated statistics, machine learning is another area that is better tackled with one of these other coding languages.

When deciding whether to use SQL, R, or Python for an analysis, consider:

- Where is the data located—in a database, a file, a website?
- What is the volume of data?
- Where is the data going—into a report, a visualization, a statistical analysis?
- Will it need to be updated or refreshed with new data? How often?
- What does your team or organization use, and how important is it to conform to existing standards?

There is no shortage of debate around which languages and tools are best for doing data analysis or data science. As with many things, there's often more than one way to accomplish an analysis. Programming languages evolve and change in popularity, and we're lucky to live and work in a time with so many good choices. SQL has been around for a long time and will likely remain popular for years to come. The ultimate goal is to use the best available tool for the job. This book will help you get the most out of SQL for data analysis, regardless of what else is in your toolkit.

SQL as Part of the Data Analysis Workflow

Now that I've explained what SQL is, discussed some of its benefits, and compared it to other languages, we'll turn to a discussion of where SQL fits in the data analysis process. Analysis work always starts with a question, which may be about how many new customers have been acquired, how sales are trending, or why some users stick around for a long time while others try a service and never return. Once the question is framed, we consider where the data originated, where the data is stored, the

1 There are some newer technologies that allow SQL queries on data stored in nonrelational sources.

analysis plan, and how the results will be presented to the audience. Figure 1-2 shows the steps in the process. Queries and analysis are the focus of this book, though I will discuss the other steps briefly in order to put the queries and analysis stage into a broader context.

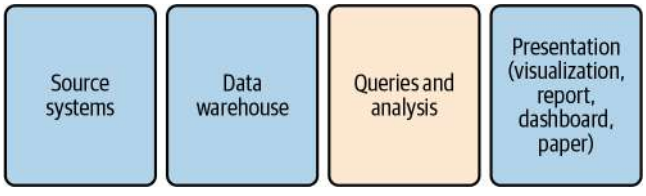


Figure 1-2. Steps in the data analysis process

First, data is generated by *source systems*, a term that includes any human or machine process that generates data of interest. Data can be generated by people by hand, such as when someone fills out a form or takes notes during a doctor's visit. Data can also be machine generated, such as when an application database records a purchase, an event-streaming system records a website click, or a marketing management tool records an email open. Source systems can generate many different types and formats of data, and Chapter 2 will discuss them, and how the type of source may impact the analysis, in more detail.

The second step is moving the data and storing it in a database for analysis. I will use the terms *data warehouse*, which is a database that consolidates data from across an organization into a central repository, and *data store*, which refers to any type of data storage system that can be queried. Other terms you might come across are *data mart*, which is typically a subset of a data warehouse, or a more narrowly focused data warehouse; and *data lake*, a term that can mean either that data resides in a file storage system or that it is stored in a database but without the degree of data transformation that is common in data warehouses. Data warehouses range from small and simple to huge and expensive. A database running on a laptop will be sufficient for you to follow along with the examples in this book. What matters is having the data you need to perform an analysis together in one place.

Usually a person or team is responsible for getting data into the data warehouse. This process is called *ETL*, or extract, transform, load. Extract pulls the data from the source system. Transform optionally changes the structure of the data, performs data quality cleaning, or aggregates the data. Load puts the data into the database. This process can also be called *ELT*, for extract, load, transform—the difference being that, rather than transformations being done before data is loaded, all the data is loaded and then transformations are performed, usually using SQL. You might also hear the terms *source* and *target* in the context of ETL. The source is where the data comes from, and the target is the destination, i.e., the database and the tables within it. Even when SQL is used to do the transforming, another language such as Python or Java is used to glue the steps together, coordinate scheduling, and raise alerts when something goes wrong. There are a number of commercial products as well as open source tools available, so teams don't have to create an ETL system entirely from scratch.

Once the data is in a database, the next step is performing queries and analysis. In this step, SQL is applied to explore, profile, clean, shape, and analyze the data. Figure 1-3 shows the general flow of the process. Exploring the data involves becoming familiar with the topic, where the data was generated, and the database tables in which it is stored. Profiling involves checking the unique values and distribution of records in the data set. Cleaning involves fixing incorrect or incomplete data, adding categorization and flags, and handling null values. Shaping is the process of arranging the data into the rows and columns needed in the result set. Finally, analyzing the data involves reviewing the output for trends, conclusions, and insights. Although this process is shown as linear, in practice it is often cyclical—for example, when shaping or analysis reveals data that should be cleaned.

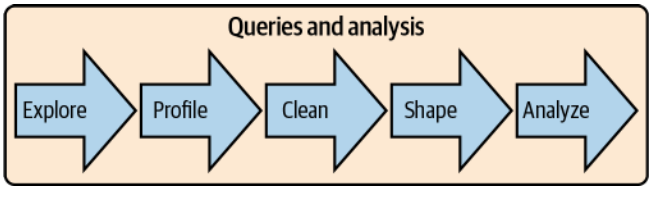


Figure 1-3. Stages within the queries and analysis step of the analysis workflow

Presentation of the data into a final output form is the last step in the overall workflow. Businesspeople won't appreciate receiving a file of SQL code; they expect you to present graphs, charts, and insights. Communication is key to having an impact with analysis, and for that we need a way to share the results with other people. At other times, you may need to apply more sophisticated statistical analysis than is possible in SQL, or you may want to feed the data into a machine learning (ML) algorithm. Fortunately, most reporting and visualization tools have SQL connectors that allow you to pull in data from entire tables or prewritten SQL queries. Statistical software and languages commonly used for ML also usually have SQL connectors.

Analysis workflows encompass a number of steps and often include multiple tools and technologies. SQL queries and analysis are at the heart of many analyses and are what we will focus on in the following chapters. Chapter 2 will discuss types of source systems and the types of data they generate. The rest of this chapter will take a look at the types of databases you are likely to encounter in your analysis journey.

Database Types and How to Work with Them

If you're working with SQL, you'll be working with databases. There is a range of database types—open source to proprietary, row-store to column-store. There are on-premises databases and cloud databases, as well as hybrid databases, where an organization runs the database software on a cloud vendor's infrastructure. There are also a number of data stores that aren't databases at all but can be queried with SQL.

Databases are not all created equal; each database type has its strengths and weaknesses when it comes to analysis work. Unlike tools used in other parts of the analysis workflow, you may not have much say in which database technology is used in your organization. Knowing the ins and outs of the database you have will help you work more efficiently and take advantage of any special SQL functions it offers. Familiarity with other types of databases will help you if you find yourself working on a project to build or migrate to a new data warehouse. You may want to install a database on your laptop for personal, small-scale projects, or get an instance of a cloud warehouse for similar reasons.

Databases and data stores have been a dynamic area of technology development since they were introduced. A few trends since the turn of the 21st century have driven the technology in ways that are really exciting for data practitioners today. First, data volumes have increased incredibly with the internet, mobile devices, and the Internet of Things (IoT). In 2020 IDC predicted that the amount of data stored globally will grow to 175 zettabytes by 2025. This scale of data is hard to even think about, and not all of it will be stored in databases for analysis. It's not uncommon for companies to have data in the scale of terabytes and petabytes these days, a scale that would have been impossible to process with the technology of the 1990s and earlier. Second, decreases in data storage and computing costs, along with the advent of the cloud,

have made it cheaper and easier for organizations to collect and store these massive amounts of data. Computer memory has gotten cheaper, meaning that large amounts of data can be loaded into memory, calculations performed, and results returned, all without reading and writing to disk, greatly increasing the speed. Third, distributed computing has allowed the breaking up of workloads across many machines. This allows a large and tunable amount of computing to be pointed to complex data tasks.

Databases and data stores have combined these technological trends in a number of different ways in order to optimize for particular types of tasks. There are two broad categories of databases that are relevant for analysis work: row-store and column-store. In the next section I'll introduce them, discuss what makes them similar to and different from each other, and talk about what all of this means as far as doing analysis with data stored in them. Finally, I'll introduce some additional types of data infrastructure beyond databases that you may encounter.

Row-Store Databases

Row-store databases—also called *transactional* databases—are designed to be efficient at processing transactions: *INSERT*s, *UPDATE*s, and *DELETE*s. Popular open source row-store databases include MySQL and Postgres. On the commercial side, Microsoft SQL Server, Oracle, and Teradata are widely used. Although they're not really optimized for analysis, for a number of years row-store databases were the only option for companies building data warehouses. Through careful tuning and schema design, these databases can be used for analytics. They are also attractive due to the low cost of open source options and because they're familiar to the database administrators who maintain them. Many organizations replicate their production database in the same technology as a first step toward building out data infrastructure. For all of these reasons, data analysts and data scientists are likely to work with data in a row-store database at some point in their career.

We think of a table as rows and columns, but data has to be serialized for storage. A query searches a hard disk for the needed data. Hard disks are organized in a series of blocks of a fixed size. Scanning the hard disk takes both time and resources, so minimizing the amount of the disk that needs to be scanned to return query results is important. Row-store databases approach this problem by serializing data in a row. Figure 1-4 shows an example of row-wise data storage. When querying, the whole row is read into memory. This approach is fast when making row-wise updates, but it's slower when making calculations across many rows if only a few columns are needed.

id	sku	type	color	size	price
1	123	tshirt	black	S	19.99
2	124	shorts	green	M	24.99

Figure 1-4. Row-wise storage, in which each row is stored together on disk

To reduce the width of tables, row-store databases are usually modeled in *third normal form*, which is a database design approach that seeks to store each piece of information only once, to avoid duplication and inconsistencies. This is efficient for transaction processing but often leads to a large number of tables in the database, each with only a few columns. To analyze such data, many joins may be required, and it can be difficult for nondevelopers to understand how all of the tables relate to each other and where a particular piece of data is stored. When doing analysis, the goal is usually denormalization, or getting all the data together in one place.

Tables typically have a *primary key* that enforces uniqueness—in other words, it prevents the database from creating more than one record for the same thing. Tables will often have an `id` column that is an auto-incrementing integer, where each new record gets the next integer after the last one inserted, or an alphanumeric value that is created by a primary key generator. There should also be a set of columns that together make the row unique; this combination of fields is called a *composite key*, or sometimes a *business key*. For example, in a table of people, the columns `first_name`, `last_name`, and `birthdate` together might make the row unique. `Social_security_id` would also be a unique identifier, in addition to the table's `person_id` column.

Tables also optionally have indexes that make looking up specific records faster and make joins involving these columns faster. Indexes store the values in the field or fields indexed as single pieces of data along with a row pointer, and since the indexes are smaller than the whole table, they are faster to scan. Usually the primary key is indexed, but other fields or groups of fields can be indexed as well. When working with row-store databases, it's useful to get to know which fields in the tables you use have indexes. Common joins can be sped up by adding indexes, so it's worth investigating whether analysis queries take a long time to run. Indexes don't come for free: they take up storage space, and they slow down loading, as new values need to be added with each insert. DBAs may not index everything that might be useful for analysis. Beyond reporting, analysis work may not be routine enough to bother with optimizing indexes either. Exploratory and complex queries often use complex join patterns, and we may throw out one approach when we figure out a new way to solve a problem.

Star schema modeling was developed in part to make row-store databases more friendly to analytic workloads. The foundations are laid out in the book *The Data*

Warehouse Toolkit,[2] which advocates modeling the data as a series of fact and dimension tables. Fact tables represent events, such as retail store transactions. Dimensions hold descriptors such as customer name and product type. Since data doesn't always fit neatly into fact and dimension categories, there's an extension called the snowflake schema in which some dimensions have dimensions of their own.

Column-Store Databases

Column-store databases took off in the early part of the 21st century, though their theoretical history goes back as far as that of row-store databases. Column-store databases store the values of a column together, rather than storing the values of a row together. This design is optimized for queries that read many records but not necessarily all the columns. Popular column-store databases include Amazon Redshift, Snowflake, and Vertica.

Column-store databases are efficient at storing large volumes of data thanks to compression. Missing values and repeating values can be represented by very small marker values instead of the full value. For example, rather than storing "United Kingdom" thousands or millions of times, a column-store database will store a surrogate value that takes up very little storage space, along with a lookup that stores the full "United Kingdom" value. Column-store databases also compress data by taking advantage of repetitions of values in sorted data. For example, the database can store the fact that the marker value for "United Kingdom" is repeated 100 times, and this takes up even less space than storing that marker 100 times.

Column-store databases do not enforce primary keys and do not have indexes. Repeated values are not problematic, thanks to compression. As a result, schemas can be tailored for analysis queries, with all the data together in one place as opposed to being in multiple tables that need to be joined. Duplicate data can easily sneak in without primary keys, however, so understanding the source of the data and quality checking are important.

Updates and deletes are expensive in most column-store databases, since data for a single row is distributed rather than stored together. For very large tables, a write-only policy may exist, so we also need to know something about how the data is generated in order to figure out which records to use. The data can also be slower to read, as it needs to be uncompressed before calculations are applied.

Column-store databases are generally the gold standard for fast analysis work. They use standard SQL (with some vendor-specific variations), and in many ways working with them is no different from working with a row-store database in terms of the queries you write. The size of the data matters, as do the computing and storage

2 Ralph Kimball and Margy Ross, *The Data Warehouse Toolkit*, 3rd ed. (Indianapolis: Wiley, 2013).

resources that have been allocated to the database. I have seen aggregations run across millions and billions of records in seconds. This does wonders for productivity.

There are a few tricks to be aware of. Since certain types of compression rely on sorting, knowing the fields that the table is sorted on and using them to filter queries improves performance. Joining tables can be slow if both tables are large.

At the end of the day, some databases will be easier or faster to work with, but there is nothing inherent in the type of database that will prevent you from performing any of the analysis in this book. As with all things, using a tool that's properly powerful for the volume of data and complexity of the task will allow you to focus on creating meaningful analysis.

Other Types of Data Infrastructure

Databases aren't the only way data can be stored, and there is an increasing variety of options for storing data needed for analysis and powering applications. File storage systems, sometimes called *data lakes*, are probably the main alternative to database warehouses. NoSQL databases and search-based data stores are alternative data storage systems that offer low latency for application development and searching log files. Although not typically part of the analysis process, they are increasingly part of organizations' data infrastructure, so I will introduce them briefly in this section as well. One interesting trend to point out is that although these newer types of infrastructure at first aimed to break away from the confines of SQL databases, many have ended up implementing some kind of SQL interface to query the data.

Hadoop, also known as HDFS (for "Hadoop distributed filesystem"), is an open source file storage system that takes advantage of the ever-falling cost of data storage and computing power, as well as distributed systems. Files are split into blocks, and Hadoop distributes them across a filesystem that is stored on nodes, or computers, in a cluster. The code to run operations is sent to the nodes, and they process the data in parallel. Hadoop's big breakthrough was to allow huge amounts of data to be stored cheaply. Many large internet companies, with massive amounts of often unstructured data, found this to be an advantage over the cost and storage limitations of traditional databases. Hadoop's early versions had two major downsides: specialized coding skills were needed to retrieve and process data since it was not SQL compatible, and execution time for the programs was often quite long. Hadoop has since matured, and various tools have been developed that allow SQL or SQL-like access to the data and speed up query times.

Other commercial and open source products have been introduced in the last few years to take advantage of cheap data storage and fast, often in-memory data processing, while offering SQL querying ability. Some of them even permit the analyst to write a single query that returns data from multiple underlying sources. This is exciting for anyone who works with large amounts of data, and it is validation that SQL is here to stay.

NoSQL is a technology that allows for data modeling that is not strictly relational. It allows for very low latency storage and retrieval, critical in many online applications. The class includes key-value pair storage and graph databases, which store in a node-edge format, and document stores. Examples of these data stores that you might hear about in your organization are Cassandra, Couchbase, DynamoDB, Memcached, Giraph, and Neo4j. Early on, NoSQL was marketed as making SQL obsolete, but the acronym has more recently been marketed as "not only SQL." For analysis purposes, using data stored in a NoSQL key-value store for analysis typically requires moving it to a more traditional SQL data warehouse, since NoSQL is not optimized for querying many records at once. Graph databases have applications such as network analysis, and analysis work may be done directly in them with special query languages. The tool landscape is always evolving, however, and perhaps someday we'll be able to analyze this data with SQL as well.

Search-based data stores include Elasticsearch and Splunk. Elasticsearch and Splunk are often used to analyze machine-generated data, such as logs. These and similar technologies have non-SQL query languages, but if you know SQL, you can often understand them. Recognizing how common SQL skills are, some data stores, such as Elasticsearch, have added SQL querying interfaces. These tools are useful and powerful for the use cases they were designed for, but they're usually not well suited to the types of analysis tasks this book is covering. As I've explained to people over the years, they are great for finding needles in haystacks. They're not as great at measuring the haystack itself.

Regardless of the type of database or other data storage technology, the trend is clear: even as data volumes grow and use cases become more complex, SQL is still the standard tool for accessing data. Its large existing user base, approachable learning curve, and power for analytical tasks mean that even technologies that try to move away from SQL come back around and accommodate it.

Conclusion

Data analysis is an exciting discipline with a range of applications for businesses and other organizations. SQL has many benefits for working with data, particularly any data stored in a database. Querying and analyzing data is part of the larger analysis workflow, and there are several types of data stores that a data scientist might expect to work with. Now that we've set the groundwork for analysis, SQL, and data stores,

the rest of the book will cover using SQL for analysis in depth. Chapter 2 focuses on data preparation, starting with an introduction to data types and then moving on to profiling, cleaning, and shaping data. Chapters 3 through 7 present applications of data analysis, focusing on time series analysis, cohort analysis, text analysis, anomaly detection, and experiment analysis. Chapter 8 covers techniques for developing complex data sets for further analysis in other tools. Finally, Chapter 9 concludes with thoughts on how types of analysis can be combined for new insights and lists some additional resources to support your analytics journey.

CHAPTER 2
Preparing Data for Analysis

Estimates of how long data scientists spend preparing their data vary, but it's safe to say that this step takes up a significant part of the time spent working with data. In 2014, the *New York Times* reported that data scientists spend from 50% to 80% of their time cleaning and wrangling their data. A 2016 survey by CrowdFlower found that data scientists spend 60% of their time cleaning and organizing data in order to prepare it for analysis or modeling work. Preparing data is such a common task that terms have sprung up to describe it, such as data munging, data wrangling, and data prep. ("Mung" is an acronym for Mash Until No Good, which I have certainly done on occasion.) Is all this data preparation work just mindless toil, or is it an important part of the process?

Data preparation is easier when a data set has a *data dictionary*, a document or repository that has clear descriptions of the fields, possible values, how the data was collected, and how it relates to other data. Unfortunately, this is frequently not the case. Documentation often isn't prioritized, even by people who see its value, or it becomes out-of-date as new fields and tables are added or the way data is populated changes. Data profiling creates many of the elements of a data dictionary, so if your organization already has a data dictionary, this is a good time to use it and contribute to it. If no data dictionary exists currently, consider starting one! This is one of the most valuable gifts you can give to your team and to your future self. An up-to-date data dictionary allows you to speed up the data-profiling process by building on profiling that's already been done rather than replicating it. It will also improve the quality of your analysis results, since you can verify that you have used fields correctly and applied appropriate filters.

Even when a data dictionary exists, you will still likely need to do data prep work as part of the analysis. In this chapter, I'll start with a review of data types you are likely to encounter. This is followed by a review of SQL query structure. Next, I will talk

about profiling the data as a way to get to know its contents and check for data quality. Then I'll talk about some data-shaping techniques that will return the columns and rows needed for further analysis. Finally, I'll walk through some useful tools for cleaning data to deal with any quality issues.

Types of Data

Data is the foundation of analysis, and all data has a database data type and also belongs to one or more categories of data. Having a firm grasp of the many forms data can take will help you be a more effective data analyst. I'll start with the database data types most frequently encountered in analysis. Then I'll move on to some conceptual groupings that can help us understand the source, quality, and possible applications of the data.

Database Data Types

Fields in database tables all have defined data types. Most databases have good documentation on the types they support, and this is a good resource for any needed detail beyond what is presented here. You don't necessarily need to be an expert on the nuances of data types to be good at analysis, but later in the book we'll encounter situations in which considering the data type is important, so this section will cover the basics. The main types of data are strings, numeric, logical, and datetime, as summarized in Table 2-1. These are based on Postgres but are similar across most major database types.

Table 2-1. A summary of common database data types

Type	Name	Description
String	CHAR / VARCHAR	Holds strings. A CHAR is always of fixed length, whereas a VARCHAR is of variable length, up to some maximum size (256 characters, for example).
	TEXT / BLOB	Holds longer strings that don't fit in a VARCHAR. Descriptions or free text entered by survey respondents might be held in these fields.
Numeric	INT / SMALLINT / BIGINT	Holds integers (whole numbers). Some databases have SMALLINT and/or BIGINT. SMALLINT can be used when the field will only hold values with a small number of digits. SMALLINT takes less memory than a regular INT. BIGINT is capable of holding numbers with more digits than an INT, but it takes up more space than an INT.
	FLOAT / DOUBLE / DECIMAL	Holds decimal numbers, sometimes with the number of decimal places specified.
Logical	BOOLEAN	Holds values of TRUE or FALSE.
	DATETIME / TIMESTAMP	Holds dates with times. Typically in a YYYY-MM-DD hh:mi:ss format, where YYYY is the four-digit year, MM is the two-digit month number, DD is the two-digit day, hh is the two-digit hour (usually 24-hour time, or values of 0 to 23), mi is the two-digit minutes, and ss is the two-digit seconds. Some databases store only timestamps without time zone, while others have specific types for timestamps with and without time zones.
	TIME	Holds times.

String data types are the most versatile. These can hold letters, numbers, and special characters, including unprintable characters like tabs and newlines. String fields can be defined to hold a fixed or variable number of characters. A CHAR field could be defined to allow only two characters to hold US state abbreviations, for example, whereas a field storing the full names of states would need to be a VARCHAR to allow a variable number of characters. Fields can be defined as TEXT, CLOB (Character Large Object), or BLOB (Binary Large Object, which can include additional data types such as images), depending on the database to hold very long strings, though since they often take up a lot of space, these data types tend to be used sparingly. When data is loaded, if strings arrive that are too big for the defined data type, they may be truncated or rejected entirely. SQL has a number of string functions that we will make use of for various analysis purposes.

Numeric data types are all the ones that store numbers, both positive and negative. Mathematical functions and operators can be applied to numeric fields. Numeric data types include the INT types as well as FLOAT, DOUBLE, and DECIMAL types that allow decimal places. Integer data types are often implemented because they use less memory than their decimal counterparts. In some databases, such as Postgres, dividing integers results in an integer, rather than a value with decimal places as you might expect. We'll discuss converting numeric data types to obtain correct results later in this chapter.

The logical data type is called BOOLEAN. It has values of TRUE and FALSE and is an efficient way to store information where these options are appropriate. Operations that compare two fields return a BOOLEAN value as a result. This data type is often used to create *flags*, fields that summarize the presence or absence of a property in the data. For example, a table storing email data might have a BOOLEAN `has_opened` field.

The datetime types include DATE, TIMESTAMP, and TIME. Date and time data should be stored in a field of one of these database types whenever possible, since SQL has a number of useful functions that operate on them. Timestamps and dates are very common in databases and are critical to many types of analysis, particularly time series analysis (covered in Chapter 3) and cohort analysis (covered in Chapter 4). Chapter 3 will discuss date and time formatting, transformations, and calculations.

Other data types, such as JSON and geographical types, are supported by some but not all databases. I won't go into detail on all of them here since they are generally beyond the scope of this book. However, they are a sign that SQL continues to evolve to tackle emerging analysis tasks.

Beyond database data types, there are a number of conceptual ways that data is categorized. These can have an impact both on how data is stored and on how we think about analyzing it. I will discuss these categorical data types next.

Structured Versus Unstructured

Data is often described as structured or unstructured, or sometimes as semistructured. Most databases were designed to handle *structured data*, where each attribute is stored in a column, and instances of each entity are represented as rows. A data model is first created, and then data is inserted according to that data model. For example, an address table might have fields for street address, city, state, and postal code. Each row would hold a particular customer's address. Each field has a data type and allows only data of that type to be entered. When structured data is inserted into a table, each field is verified to ensure it conforms to the correct data type. Structured data is easy to query with SQL.

Unstructured data is the opposite of structured data. There is no predetermined structure, data model, or data types. Unstructured data is often the "everything else" that isn't database data. Documents, emails, and web pages are unstructured. Photos, images, videos, and audio files are also examples of unstructured data. They don't fit into the traditional data types, and thus they are more difficult for relational databases to store efficiently and for SQL to query. Unstructured data is often stored outside of relational databases as a result. This allows data to be loaded quickly, but lack of data validation can result in low data quality. As we saw in Chapter 1, the technology continues to evolve, and new tools are being developed to allow SQL querying of many types of unstructured data.

Semistructured data falls in between these two categories. Much "unstructured" data has some structure that we can make use of. For example, emails have from and to email addresses, subject lines, body text, and sent timestamps that can be stored separately in a data model with those fields. Metadata, or data about data, can be extracted from other file types and stored for analysis. For example, music audio files might be tagged with artist, song name, genre, and duration. Generally, the structured parts of semistructured data can be queried with SQL, and SQL can often be used to parse or otherwise extract structured data for further querying. We'll see some applications of this in the discussion of text analysis in Chapter 5.

Quantitative Versus Qualitative Data

Quantitative data is numeric. It measures people, things, and events. Quantitative data can include descriptors, such as customer information, product type, or device configurations, but it also comes with numeric information such as price, quantity, or visit duration. Counts, sums, average, or other numeric functions are applied to the data. Quantitative data is often machine generated these days, but it doesn't need to be. Height, weight, and blood pressure recorded on a paper patient intake form are quantitative, as are student quiz scores typed into a spreadsheet by a teacher.

Qualitative data is usually text based and includes opinions, feelings, and descriptions that aren't strictly quantitative. Temperature and humidity levels are quantitative, while descriptors like "hot and humid" are qualitative. The price a customer paid for a product is quantitative; whether they like or dislike it is qualitative. Survey feedback, customer support inquiries, and social media posts are qualitative. There are whole professions that deal with qualitative data. In a data analysis context, we usually try to quantify the qualitative. One technique for this is to extract keywords or phrases and count their occurrences. We'll look at this in more detail when we delve into text analysis in Chapter 5. Another technique is sentiment analysis, in which the structure of language is used to interpret the meaning of the words used, in addition to their frequency. Sentences or other bodies of text can be scored for their level of positivity or negativity, and then counts or averages are used to derive insights that would be hard to summarize otherwise. There have been exciting advances in the field of natural language processing, or NLP, though much of this work is done with tools such as Python.

First-, Second-, and Third-Party Data

First-party data is collected by the organization itself. This can be done through server logs, databases that keep track of transactions and customer information, or other systems that are built and controlled by the organization and generate data of interest for analysis. Since the systems were created in-house, finding the people who built them and learning about how the data is generated is usually possible. Data analysts may also be able to influence or have control over how certain pieces of data are created and stored, particularly when bugs are responsible for poor data quality.

Second-party data comes from vendors that provide a service or perform a business function on the organization's behalf. These are often software as a service (SaaS) products; common examples are CRM, email and marketing automation tools, ecommerce-enabling software, and web and mobile interaction trackers. The data is similar to first-party data since it is about the organization itself, created by its employees and customers. However, both the code that generates and stores the data and the data model are controlled externally, and the data analyst typically has little influence over these aspects. Second-party data is increasingly imported into an organization's data warehouse for analysis. This can be accomplished with custom code or ETL connectors, or with SaaS vendors that offer data integration.

Many SaaS vendors provide some reporting capabilities, so the question may arise of whether to bother copying the data to a data warehouse. The department that interacts with a tool may find that reporting sufficient, such as a customer service department that reports on time to resolve issues and agent productivity from within its helpdesk software. On the other hand, customer service interactions might be an important input to a customer retention model, which would require integrating that data into a data store with sales and cancellation data. Here's a good rule of thumb when deciding whether to import data from a particular data source: if the data will create value when combined with data from other systems, import it; if not, wait until there is a stronger case before doing the work.

Third-party data may be purchased or obtained from free sources such as those published by governments. Unless the data has been collected specifically on behalf of the organization, data teams usually have little control over the format, frequency, and data quality. This data often lacks the granularity of first- and second-party data. For example, most third-party sources do not have user-level data, and instead data might be joined with first-party data at the postal code or city level, or at a higher level. Third-party data can have unique and useful information, however, such as aggregate spending patterns, demographics, and market trends that would be very expensive or impossible to collect otherwise.

Sparse Data

Sparse data occurs when there is a small amount of information within a larger set of empty or unimportant information. Sparse data might show up as many nulls and only a few values in a particular column. Null, different from a value of 0, is the *absence* of data; that will be covered later in the section on data cleaning. Sparse data can occur when events are rare, such as software errors or purchases of products in the long tail of a product catalog. It can also occur in the early days of a feature or product launch, when only testers or beta customers have access. JSON is one approach that has been developed to deal with sparse data from a writing and storage perspective, as it stores only the data that is present and omits the rest. This is in contrast to a row-store database, which has to hold memory for a field even if there is no value in it.

Sparse data can be problematic for analysis. When events are rare, trends aren't necessarily meaningful, and correlations are hard to distinguish from chance fluctuations. It's worth profiling your data, as discussed later in this chapter, to understand if and where your data is sparse. Some options are to group infrequent events or items into categories that are more common, exclude the sparse data or time period from

the analysis entirely, or show descriptive statistics along with cautionary explanations that the trends are not necessarily meaningful.

There are a number of different types of data and a variety of ways that data is described, many of which are overlapping or not mutually exclusive. Familiarity with these types is useful not only in writing good SQL but also for deciding how to analyze the data in appropriate ways. You may not always know the data types in advance, which is why data profiling is so critical. Before we get to that, and to our first code examples, I'll give a brief review of SQL query structure.

SQL Query Structure

SQL queries have common clauses and syntax, although these can be combined in a nearly infinite number of ways to achieve analysis goals. This book assumes you have some prior knowledge of SQL, but I'll review the basics here so that we have a common foundation for the code examples to come.

The *SELECT* clause determines the columns that will be returned by the query. One column will be returned for each expression within the *SELECT* clause, and expressions are separated by commas. An expression can be a field from the table, an aggregation such as a `sum`, or any number of calculations, such as CASE statements, type conversions, and various functions that will be discussed later in this chapter and throughout the book.

The *FROM* clause determines the tables from which the expressions in the *SELECT* clause are derived. A "table" can be a database table, a view (a type of saved query that otherwise functions like a table), or a subquery. A subquery is itself a query, wrapped in parentheses, and the result is treated like any other table by the query that references it. A query can reference multiple tables in the *FROM* clause, though they must use one of the *JOIN* types along with a condition that specifies how the tables relate. The *JOIN* condition usually specifies an equality between fields in each table, such as `orders.customer_id = customers.customer_id`. *JOIN* conditions can include multiple fields and can also specify inequalities or ranges of values, such as ranges of dates. We'll see a variety of *JOIN* conditions that achieve specific analysis goals throughout the book. An *INNER JOIN* returns all records that match in both tables. A *LEFT JOIN* returns all records from the first table, but only those records from the second table that match. A *RIGHT JOIN* returns all records from the second table, but only those records from the first table that match. A *FULL OUTER JOIN* returns all records from both tables. A Cartesian *JOIN* can result when each record in the first table matches more than one record in the second table. Cartesian *JOIN*s should generally be avoided, though there are some specific use cases, such as generating data to fill in a time series, in which we will use them intentionally. Finally, tables in the *FROM* clause can be *aliased*, or given a shorter name of one or more letters that can

be referenced in other clauses in the query. Aliases save query writers from having to type out long table names repeatedly, and they make queries easier to read.

While both *LEFT JOIN* and *RIGHT JOIN* can be used in the same query, it's much easier to keep track of your logic when you stick with only one or the other. In practice, *LEFT JOIN* is much more commonly used than *RIGHT JOIN*.

The *WHERE* clause specifies restrictions or filters that are needed to exclude or remove rows from the result set. *WHERE* is optional.

The *GROUP BY* clause is required when the *SELECT* clause contains aggregations and at least one nonaggregated field. An easy way to remember what should go in the *GROUP BY* clause is that it should have every field that is not part of an aggregation. In most databases, there are two ways to list the *GROUP BY* fields: either by field name or by position, such as 1, 2, 3, and so on. Some people prefer to use the field name notation, and SQL Server requires this. I prefer the position notation, particularly when the *GROUP BY* fields contain complex expressions or when I'm doing a lot of iteration. This book will typically use the position notation.

How Not to Kill Your Database: LIMIT and Sampling

Database tables can be very large, containing millions or billions of records. Querying across all of these records can cause problems at the least and crash databases at the worst. To avoid receiving cranky calls from database administrators or getting locked out, it's a good idea to limit the results returned during profiling or while testing queries. *LIMIT* clauses and sampling are two techniques that should be part of your toolbox.

LIMIT is added as the last line of the query, or subquery, and can take any positive integer value:

```
SELECT column_a, column_b
FROM table
LIMIT 1000
;
```

When used in a subquery, the limit will be applied at that step, and only the restricted result set will be evaluated by the outer query:

```
SELECT...
FROM
(
      SELECT column_a, column_b, sum(sales) as total_sales
      FROM table
      GROUP BY 1,2
      LIMIT 1000
```

```
) a
;
```

SQL Server does not support the *LIMIT* clause, but a similar result can be obtained using `top`:

```
SELECT top 1000
column_a, column_b
FROM table
;
```

Sampling can be accomplished by using a function on an ID field that has a random distribution of digits at the beginning or end. The modulus or `mod` function returns the remainder when one integer is divided by another. If the ID field is an integer, `mod` can be used to find the last one, two, or more digits and filter on the result:

```
WHERE mod(integer_order_id,100) = 6
```

This will return every order whose last two digits are 06, which should be about 1% of the total. If the field is alphanumeric, you can use a `right()` function to find a certain number of digits at the end:

```
WHERE right(alphanum_order_id,1) = 'B'
```

This will return every order with a last digit of B, which will be about 3% of the total if all letters and numbers are equally common, an assumption worth validating.

Limiting the result set also makes your work faster, but be aware that subsets of data might not contain all of the variations in values and edge cases that exist in the full data set. Remember to remove the *LIMIT* or sampling before running your final analysis or report with your query, or you'll end up with funny results!

That covers the basics of SQL query structure. Chapter 8 will go into additional detail on each of these clauses, a few additional ones that are less commonly encountered but appear in this book, and the order in which each clause is evaluated. Now that we have this foundation, we can turn to one of the most important parts of the analysis process: data profiling.

Profiling: Distributions

Profiling is the first thing I do when I start working with any new data set. I look at how the data is arranged into schemas and tables. I look at the table names to get familiar with the topics covered, such as customers, orders, or visits. I check out the column names in a few tables and start to construct a mental model of how the tables relate to one another. For example, the tables might include an `order_detail` table with line-item breakouts that relate to the `order` table via an `order_id`, while the `order` table relates to the `customer` table via a `customer_id`. If there is a data dictionary, I review that and compare it to the data I see in a sample of rows.

The tables generally represent the operations of an organization, or some subset of the operations, so I think about what domain or domains are covered, such as ecommerce, marketing, or product interactions. Working with data is easier when we have knowledge of how the data was generated. Profiling can provide clues about this, or about what questions to ask of the source, or of people inside or outside the organization responsible for the collection or generation of the data. Even when you collect the data yourself, profiling is useful.

Another detail I check for is how history is represented, if at all. Data sets that are replicas of production databases may not contain previous values for customer addresses or order statuses, for example, whereas a well-constructed data warehouse may have daily snapshots of changing data fields.

Profiling data is related to the concept of *exploratory data analysis*, or EDA, named by John Tukey. In his book of that name,[1] Tukey describes how to analyze data sets by computing various summaries and visualizing the results. He includes techniques for looking at distributions of data, including stem-and-leaf plots, box plots, and histograms.

After checking a few samples of data, I start looking at distributions. Distributions allow me to understand the range of values that exist in the data and how often they occur, whether there are nulls, and whether negative values exist alongside positive ones. Distributions can be created with continuous or categorical data and are also called frequencies. In this section, we'll look at how to create histograms, how binning can help us understand the distribution of continuous values, and how to use n-tiles to get more precise about distributions.

Histograms and Frequencies

One of the best ways to get to know a data set, and to know particular fields within the data set, is to check the frequency of values in each field. Frequency checks are also useful whenever you have a question about whether certain values are possible or if you spot an unexpected value and want to know how commonly it occurs. Frequency checks can be done on any data type, including strings, numerics, dates, and booleans. Frequency queries are a great way to detect sparse data as well.

The query is straightforward. The number of rows can be found with `count(*)`, and the profiled field is in the *GROUP BY*. For example, we can check the frequency of each type of `fruit` in a fictional `fruit_inventory` table:

1 John W. Tukey, *Exploratory Data Analysis* (Reading, MA: Addison-Wesley, 1977).

```
SELECT fruit, count(*) as quantity
FROM fruit_inventory
GROUP BY 1
;
```

When using `count`, it's worth taking a minute to consider whether there might be any duplicate records in the data set. You can use `count(*)` when you want the number of records, but use `count distinct` to find out how many unique items there are.

A *frequency plot* is a way to visualize the number of times something occurs in the data set. The field being profiled is usually plotted on the x-axis, with the count of observations on the y-axis. Figure 2-1 shows an example of plotting the frequency of fruit from our query. Frequency graphs can also be drawn horizontally, which accommodates long value names well. Notice that this is categorical data without any inherent order.

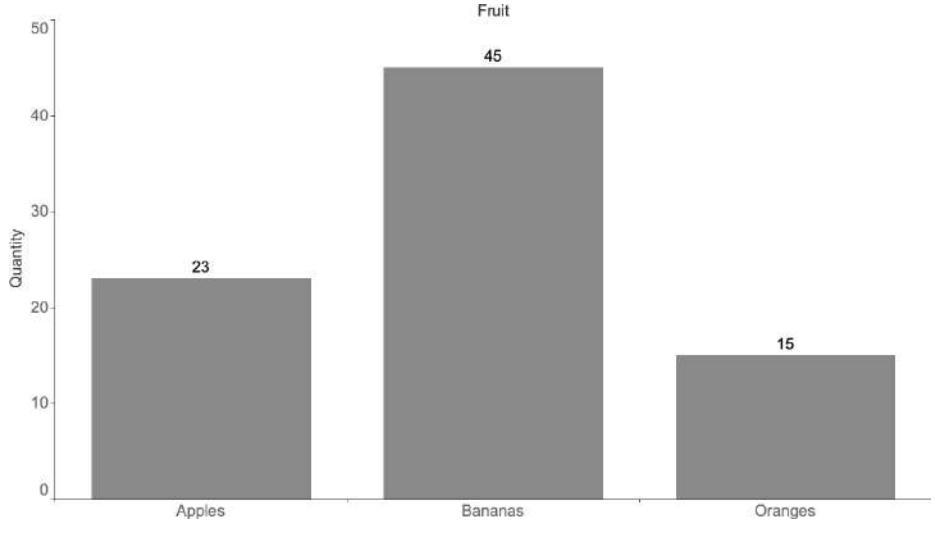


Figure 2-1. Frequency plot of fruit inventory

A *histogram* is a way to visualize the distribution of numerical values in a data set and will be familiar to those with a statistics background. A basic histogram might show the distribution of ages across a group of customers. Imagine that we have a `customers` table that contains names, registration date, age, and other attributes. To create a histogram by age, *GROUP BY* the numerical `age` field and `count customer_id`:

```
SELECT age, count(customer_id) as customers
FROM customers
GROUP BY 1
;
```

The results of our hypothetical age distribution are graphed in Figure 2-2.

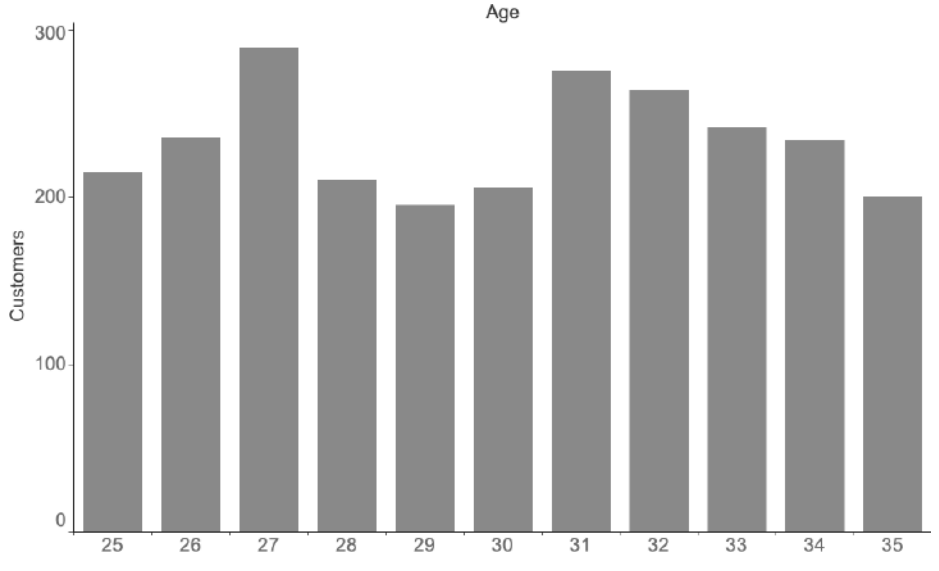


Figure 2-2. Customers by age

Another technique I've used repeatedly and that has become the basis for one of my favorite interview questions involves an aggregation followed by a frequency count. I give candidates a hypothetical table called `orders`, which has a date, customer identifier, order identifier, and an amount, and then ask them to write a SQL query that returns the distribution of orders per customer. This can't be solved with a simple query; it requires an intermediate aggregation step, which can be accomplished with a subquery. First, `count` the number of orders placed by each `customer_id` in the subquery. The outer query uses the number of `orders` as a category and `counts` the number of customers:

```
SELECT orders, count(*) as num_customers
FROM
(
    SELECT customer_id, count(order_id) as orders
    FROM orders
    GROUP BY 1
) a
GROUP BY 1
;
```

This type of profiling can be applied whenever you need to see how frequently certain entities or attributes appear in the data. In these examples, `count` has been used, but the other basic aggregations (`sum`, `avg`, `min`, and `max`) can be used to create histograms as well. For instance, we might want to profile customers by the `sum` of all their orders, their `avg` order size, their `min` order date, or their `max` (most recent) order date.

Binning

Binning is useful when working with continuous values. Rather than the number of observations or records for each value being counted, ranges of values are grouped together, and these groups are called *bins* or *buckets*. The number of records that fall into each interval is then counted. Bins can be variable in size or have a fixed size, depending on whether your goal is to group the data into bins that have particular meaning for the organization, are roughly equal width, or contain roughly equal numbers of records. Bins can be created with CASE statements, rounding, and logarithms.

A CASE statement allows for conditional logic to be evaluated. These statements are very flexible, and we will come back to them throughout the book, applying them to data profiling, cleaning, text analysis, and more. The basic structure of a CASE statement is:

```
case when condition1 then return_value_1
     when condition2 then return_value_2
     ...
     else return_value_default
     end
```

The WHEN condition can be an equality, inequality, or other logical condition. The THEN return value can be a constant, an expression, or a field in the table. Any number of conditions can be included, but the statement will stop executing and return the result the first time a condition evaluates to TRUE. ELSE tells the database what to use as a default value if no matches are found and can also be a constant or field. ELSE is optional, and if it is not included, any nonmatches will return null. CASE statements can also be nested so that the return value is another CASE statement.

The return values following THEN must all be the same data type (strings, numeric, BOOLEAN, etc.), or else you'll get an error. Consider casting to a common data type such as string if you encounter this.

A CASE statement is a flexible way to control the number of bins, the range of values that fall into each bin, and how the bins are named. I find them particularly useful when there is a long tail of very small or very large values that I want to group

together rather than have empty bins in part of the distribution. Certain ranges of values have a business meaning that needs to be re-created in the data. Many B2B companies separate their customers into "enterprise" and "SMB" (small- and medium-sized businesses) categories based on number of employees or revenue, because their buying patterns are different. As an example, imagine we are considering discounted shipping offers and we want to know how many customers will be affected. We can group `order_amount` into three buckets using a CASE statement:

```
SELECT
case when order_amount <= 100 then 'up to 100'
     when order_amount <= 500 then '100 - 500'
     else '500+' end as amount_bin
,case when order_amount <= 100 then 'small'
      when order_amount <= 500 then 'medium'
      else 'large' end as amount_category
,count(customer_id) as customers
FROM orders
GROUP BY 1,2
;
```

Arbitrary-sized bins can be useful, but at other times bins of fixed size are more appropriate for the analysis. Fixed-size bins can be accomplished in a few ways, including with rounding, logarithms, and n-tiles. To create equal-width bins, rounding is useful. Rounding reduces the precision of the values, and we usually think about rounding as reducing the number of decimal places or removing them altogether by rounding to the nearest integer. The `round` function takes the form:

```
round(value,number_of_decimal_places)
```

The number of decimal places can also be a negative number, allowing this function to round to the nearest tens, hundreds, thousands, and so on. Table 2-2 demonstrates the results of rounding with arguments ranging from –3 to 2.

Table 2-2. The number 123,456.789 rounded with various decimal places

Decimal places	Formula	Result
2	round(123456.789,2)	123456.79
1	round(123456.789,1)	123456.8
0	round(123456.789,0)	123457
-1	round(123456.789,-1)	123460
-2	round(123456.789,-2)	123500
-3	round(123456.789,-3)	123000

```
SELECT round(sales,-1) as bin
,count(customer_id) as customers
FROM table
GROUP BY 1
;
```

Logarithms are another way to create bins, particularly in data sets in which the largest values are orders of magnitude greater than the smallest values. The distribution of household wealth, the number of website visitors across different properties on the internet, and the shaking force of earthquakes are all examples of phenomena that have this property. While they don't create bins of equal width, logarithms create bins that increase in size with a useful pattern. To refresh your memory, a logarithm is the exponent to which 10 must be raised to produce that number:

$$\log(number) = exponent$$

In this case, 10 is called the base, and this is usually the default implementation in databases, but technically the base can be any number. Table 2-3 shows the logarithms for several powers of 10.

Table 2-3. Results of `log` function on powers of 10

Formula	Result
log(1)	0
log(10)	1
log(100)	2
log(1000)	3
log(10000)	4

In SQL, the `log` function returns the logarithm of its argument, which can be a constant or a field:

```
SELECT log(sales) as bin
,count(customer_id) as customers
FROM table
GROUP BY 1
;
```

The `log` function can be used on any positive value, not just multiples of 10. However, the logarithm function does not work when values can be less than or equal to 0; it will return null or an error, depending on the database.

n-Tiles

You're probably familiar with the *median*, or middle value, of a data set. This is the 50th percentile value. Half of the values are larger than the median, and the other half are smaller. With quartiles, we fill in the 25th and 75th percentile values. A quarter of the values are smaller and three quarters are larger for the 25th percentile; three quarters are smaller and one quarter are larger at the 75th percentile. Deciles break the data set into 10 equal parts. Making this concept generic, *n-tiles* allow us to calculate any percentile of the data set: 27th percentile, 50.5th percentile, and so on.

Window Functions

The n-tiles functions are part of a group of SQL functions called window or analytic functions. Unlike most SQL functions, which can operate only on the current row of data, window functions perform calculations that span multiple rows. Window functions have special syntax that includes the function name and an *OVER* clause that is used to determine the rows on which to operate and the ordering of those rows. The general format of a window function is:

```
function(field_name) over (partition by field_name order by field_name)
```

The function can be any of the normal aggregations (`count`, `sum`, `avg`, `min`, `max`) as well as a number of special functions, including `rank`, `first_value`, and `ntile`. The *PARTITION BY* clause can include zero or more fields. When no fields are specified, the function operates over the entire table, but when one or more fields are specified, the function will operate only on that section of rows. For example, we might *PARTITION BY* a `customer_id` to perform calculations about all of the records per customer, restarting the calculation for each customer. The *ORDER BY* clause determines the ordering of the rows for functions that rely on this; for example, to `rank` customers, we need to specify a field by which to order them, such as number of orders. All of the major database types have window functions, except for versions of MySQL prior to 8.0.2. We will see these useful functions throughout the book, along with additional explanations of how they work and how to set up the arguments correctly.

Many databases have a `median` function built in but rely on more generic n-tile functions for the rest. These functions are window functions, computing across a range of rows to return a value for a single row. They take an argument that specifies the number of bins to split the data into and, optionally, a *PARTITION BY* and/or an *ORDER BY* clause:

```
ntile(num_bins) over (partition by... order by...)
```

As an example, imagine we had 12 transactions with `order_amounts` of $19.99, $9.99, $59.99, $11.99, $23.49, $55.98, $12.99, $99.99, $14.99, $34.99, $4.99, and $89.99. Performing an `ntile` calculation with 10 bins sorts each `order_amount` and assigns a bin from 1 to 10:

```
order_amount  ntile
------------  -----
4.99          1
9.99          1
11.99         2
12.99         2
14.99         3
19.99         4
23.49         5
34.99         6
```

```
55.98         7
59.99         8
89.99         9
99.99         10
```

This can be used to bin records in practice by first calculating the `ntile` of each row in a subquery and then wrapping it in an outer query that uses `min` and `max` to find the upper and lower boundaries of the value range:

```
SELECT ntile
,min(order_amount) as lower_bound
,max(order_amount) as upper_bound
,count(order_id) as orders
FROM
(
    SELECT customer_id, order_id, order_amount
    ,ntile(10) over (order by order_amount) as ntile
    FROM orders
) a
GROUP BY 1
;
```

A related function is `percent_rank`. Instead of returning the bins that the data falls into, `percent_rank` returns the percentile. It takes no argument but requires parentheses and optionally takes a *PARTITION BY* and/or an *ORDER BY* clause:

```
percent_rank() over (partition by... order by...)
```

While not as useful as `ntile` for binning, `percent_rank` can be used to create a continuous distribution, or it can be used as an output itself for reporting or further analysis. Both `ntile` and `percent_rank` can be expensive to compute over large data sets, since they require sorting all the rows. Filtering the table to only the data set you need helps. Some databases have implemented approximate versions of the functions that are faster to compute and generally return high-quality results if absolute precision is not required. We will look at additional uses for n-tiles in the discussion of anomaly detection in Chapter 6.

In many contexts, there is no single correct or objectively best way to look at distributions of data. There is significant leeway for analysts to use the preceding techniques to understand data and present it to others. However, data scientists need to use judgment and must bring their ethical radar along whenever sharing distributions of sensitive data.

Profiling: Data Quality

Data quality is absolutely critical when it comes to creating good analysis. Although this may seem obvious, it has been one of the hardest lessons I've learned in my years of working with data. It's easy to get overly focused on the mechanics of processing

the data, finding clever query techniques and just the right visualization, only to have stakeholders ignore all of that and point out the one data inconsistency. Ensuring data quality can be one of the hardest and most frustrating parts of analysis. The saying "garbage in, garbage out" captures only part of the problem. Good ingredients in plus incorrect assumptions can also lead to garbage out.

Comparing data against ground truth, or what is otherwise known to be true, is ideal though not always possible. For example, if you are working with a replica of a production database, you could compare the row counts in each system to verify that all rows arrived in the replica database. In other cases, you might know the dollar value and count of sales in a particular month and thus can query for this information in the database to make sure the `sum` of sales and `count` of records match. Often the difference between your query results and the expected value comes down to whether you applied the correct filters, such as excluding cancelled orders or test accounts; how you handled nulls and spelling anomalies; and whether you set up correct *JOIN* conditions between tables.

Profiling is a way to uncover data quality issues early on, before they negatively impact results and conclusions drawn from the data. Profiling reveals nulls, categorical codings that need to be deciphered, fields with multiple values that need to be parsed, and unusual datetime formats. Profiling can also uncover gaps and step changes in the data that have resulted from tracking changes or outages. Data is rarely perfect, and it's often only through its use in analysis that data quality issues are uncovered.

Detecting Duplicates

A *duplicate* is when you have two (or more) rows with the same information. Duplicates can exist for any number of reasons. A mistake might have been made during data entry, if there is some manual step. A tracking call might have fired twice. A processing step might have run multiple times. You might have created it accidentally with a hidden many-to-many *JOIN*. However they come to be, duplicates can really throw a wrench in your analysis. I can recall times early in my career when I thought I had a great finding, only to have a product manager point out that my sales figure was twice the actual sales. It's embarrassing, it erodes trust, and it requires rework and sometimes painstaking reviews of the code to find the problem. I've learned to check for duplicates as I go.

Fortunately, it's relatively easy to find duplicates in our data. One way is to inspect a sample, with all columns ordered:

```
SELECT column_a, column_b, column_c...
FROM table
ORDER BY 1,2,3...
;
```

This will reveal whether the data is full of duplicates, for example, when looking at a brand-new data set, when you suspect that a process is generating duplicates, or after a possible Cartesian *JOIN*. If there are only a few duplicates, they might not show up in the sample. And scrolling through data to try to spot duplicates is taxing on your eyes and brain. A more systematic way to find duplicates is to *SELECT* the columns and then `count` the rows (this might look familiar from the discussion of histograms!):

```
SELECT count(*)
FROM
(
    SELECT column_a, column_b, column_c...
    , count(*) as records
    FROM...
    GROUP BY 1,2,3...
) a
WHERE records > 1
;
```

This will tell you whether there are any cases of duplicates. If the query returns 0, you're good to go. For more detail, you can list out the number of records (2, 3, 4, etc.):

```
SELECT records, count(*)
FROM
(
    SELECT column_a, column_b, column_c..., count(*) as records
    FROM...
    GROUP BY 1,2,3...
) a
WHERE records > 1
GROUP BY 1
;
```

As an alternative to a subquery, you can use a *HAVING* clause and keep everything in a single main query. Since it is evaluated after the aggregation and *GROUP BY*, *HAVING* can be used to filter on the aggregation value:

```
SELECT column_a, column_b, column_c..., count(*) as records
FROM...
GROUP BY 1,2,3...
HAVING count(*) > 1
;
```

I prefer to use subqueries, because I find that they're a useful way to organize my logic. Chapter 8 will discuss order of evaluation and strategies for keeping your SQL queries organized.

For full detail on which records have duplicates, you can list out all the fields and then use this information to chase down which records are problematic:

```
SELECT *
FROM
(
    SELECT column_a, column_b, column_c..., count(*) as records
    FROM...
    GROUP BY 1,2,3...
) a
WHERE records = 2
;
```

Detecting duplicates is one thing; figuring out what to do about them is another. It's almost always useful to understand why duplicates are occurring and, if possible, fix the problem upstream. Can a data process be improved to reduce or remove duplication? Is there an error in an ETL process? Have you failed to account for a one-to-many relationship in a *JOIN*? Next, we'll turn to some options for handling and removing duplicates with SQL.

Deduplication with GROUP BY and DISTINCT

Duplicates happen, and they're not always a result of bad data. For example, imagine we want to find a list of all the customers who have successfully completed a transaction so we can send them a coupon for their next order. We might *JOIN* the `customers` table to the `transactions` table, which would restrict the records returned to only those customers that appear in the `transactions` table:

```
SELECT a.customer_id, a.customer_name, a.customer_email
FROM customers a
JOIN transactions b on a.customer_id = b.customer_id
;
```

This will return a row for each customer for each transaction, however, and there are hopefully at least a few customers who have transacted more than once. We have accidentally created duplicates, not because there is any underlying data quality problem but because we haven't taken care to avoid duplication in the results. Fortunately, there are several ways to avoid this with SQL. One way to remove duplicates is to use the keyword *DISTINCT*:

```
SELECT distinct a.customer_id, a.customer_name, a.customer_email
FROM customers a
JOIN transactions b on a.customer_id = b.customer_id
;
```

Another option is to use a *GROUP BY*, which, although typically seen in connection with an aggregation, will also deduplicate in the same way as *DISTINCT*. I remember the first time I saw a colleague use *GROUP BY* without an aggregation dedupe—I

didn't even realize it was possible. I find it somewhat less intuitive than *DISTINCT*, but the result is the same:

```
SELECT a.customer_id, a.customer_name, a.customer_email
FROM customers a
JOIN transactions b on a.customer_id = b.customer_id
GROUP BY 1,2,3
;
```

Another useful technique is to perform an aggregation that returns one row per entity. Although technically not deduping, it has a similar effect. For example, if we have a number of transactions by the same customer and need to return one record per customer, we could find the `min` (first) and/or the `max` (most recent) `transaction_date`:

```
SELECT customer_id
,min(transaction_date) as first_transaction_date
,max(transaction_date) as last_transaction_date
,count(*) as total_orders
FROM table
GROUP BY customer_id
;
```

Duplicate data, or data that contains multiple records per entity even if they technically are not duplicates, is one of the most common reasons for incorrect query results. You can suspect duplicates as the cause if all of a sudden the number of customers or total sales returned by a query is many times greater than what you were expecting. Fortunately, there are several techniques that can be applied to prevent this from occurring.

Another common problem is missing data, which we'll turn to next.

Preparing: Data Cleaning

Profiling often reveals where changes can make the data more useful for analysis. Some of the steps are CASE transformations, adjusting for null, and changing data types.

Cleaning Data with CASE Transformations

CASE statements can be used to perform a variety of cleaning, enrichment, and summarization tasks. Sometimes the data exists and is accurate, but it would be more useful for analysis if values were standardized or grouped into categories. The structure of CASE statements was presented earlier in this chapter, in the section on binning.

Nonstandard values occur for a variety of reasons. Values might come from different systems with slightly different lists of choices, system code might have changed,

options might have been presented to the customer in different languages, or the customer might have been able to fill out the value rather than pick from a list.

Imagine a field containing information about the gender of a person. Values indicating a female person exist as "F," "female," and "femme." We can standardize the values like this:

```
CASE when gender = 'F' then 'Female'
     when gender = 'female' then 'Female'
     when gender = 'femme' then 'Female'
     else gender
     end as gender_cleaned
```

CASE statements can also be used to add categorization or enrichment that does not exist in the original data. As an example, many organizations use a Net Promoter Score, or NPS, to monitor customer sentiment. NPS surveys ask respondents to rate, on a scale of 0 to 10, how likely they are to recommend a company or product to a friend or colleague. Scores of 0 to 6 are considered detractors, 7 and 8 are passive, and 9 and 10 are promoters. The final score is calculated by subtracting the percentage of detractors from the percentage of promoters. Survey result data sets usually include optional free text comments and are sometimes enriched with information the organization knows about the person surveyed. Given a data set of NPS survey responses, the first step is to group the responses into the categories of detractor, passive, and promoter:

```
SELECT response_id
,likelihood
,case when likelihood <= 6 then 'Detractor'
      when likelihood <= 8 then 'Passive'
      else 'Promoter'
     end as response_type
FROM nps_responses
;
```

Note that the data type can differ between the field being evaluated and the return data type. In this case, we are checking an integer and returning a string. Listing out all the values with an IN list is also an option. The IN operator allows you to specify a list of items rather than having to write an equality for each one separately. It is useful when the input isn't continuous or when values in order shouldn't be grouped together:

```
case when likelihood in (0,1,2,3,4,5,6) then 'Detractor'
     when likelihood in (7,8) then 'Passive'
     when likelihood in (9,10) then 'Promoter'
     end as response_type
```

CASE statements can consider multiple columns and can contain AND/OR logic. They can also be nested, though often this can be avoided with AND/OR logic:

```
case when likelihood <= 6
          and country = 'US'
          and high_value = true
          then 'US high value detractor'
     when likelihood >= 9
          and (country in ('CA','JP')
               or high_value = true
              )
          then 'some other label'
     ... end
```

Alternatives for Cleaning Data

Cleaning or enriching data with a CASE statement works well as long as there is a relatively short list of variations, you can find them all in the data, and the list of values isn't expected to change. For longer lists and ones that change frequently, a lookup table can be a better option. A lookup table exists in the database and is either static or populated with code that checks for new values periodically. The query will *JOIN* to the lookup table to get the cleaned data. In this way, the cleaned values can be maintained outside your code and used by many queries, without your having to worry about maintaining consistency between them. An example of this might be a lookup table that maps state abbreviations to full state names. In my own work, I often start with a CASE statement and create a lookup table only after the list becomes unruly, or once it's clear that my team or I will need to use this cleaning step repeatedly.

Of course, it's worth investigating whether the data can be cleaned upstream. I once started with a CASE statement of 5 or so lines that grew to 10 lines and then eventually to more than 100 lines, at which point the list was unruly and difficult to maintain. The insights were valuable enough that I was able to convince engineers to change the tracking code and send the meaningful categorizations in the data stream in the first place.

Another useful thing you can do with CASE statements is to create flags indicating whether a certain value is present, without returning the actual value. This can be useful during profiling for understanding how common the existence of a particular attribute is. Another use for flagging is during preparation of a data set for statistical analysis. In this case, a flag is also known as a dummy variable, taking a value of 0 or 1 and indicating the presence or absence of some qualitative variable. For example, we can create `is_female` and `is_promoter` flags with CASE statements on `gender` and `likelihood` (to recommend) fields:

```
SELECT customer_id
,case when gender = 'F' then 1 else 0 end as is_female
,case when likelihood in (9,10) then 1 else 0 end as is_promoter
```

```
FROM ...
;
```

If you are working with a data set that has multiple rows per entity, such as with line items in an order, you can flatten the data with a CASE statement wrapped in an aggregate and turn it into a flag at the same time by using 1 and 0 as the return value. We saw previously that a BOOLEAN data type is often used to create flags (fields that represent the presence or absence of some attribute). Here, 1 is substituted for TRUE and 0 is substituted for FALSE so that a `max` aggregation can be applied. The way this works is that for each customer, the CASE statement returns 1 for any row with a fruit type of "apple." Then `max` is evaluated and will return the largest value from any of the rows. As long as a customer bought an apple at least once, the flag will be 1; if not, it will be 0:

```
SELECT customer_id
,max(case when fruit = 'apple' then 1
          else 0
          end) as bought_apples
,max(case when fruit = 'orange' then 1
          else 0
          end) as bought_oranges
FROM ...
GROUP BY 1
;
```

You can also construct more complex conditions for flags, such as requiring a threshold or amount of something before labeling with a value of 1:

```
SELECT customer_id
,max(case when fruit = 'apple' and quantity > 5 then 1
          else 0
          end) as loves_apples
,max(case when fruit = 'orange' and quantity > 5 then 1
          else 0
          end) as loves_oranges
FROM ...
GROUP BY 1
;
```

CASE statements are powerful, and as we saw, they can be used to clean, enrich, and flag or add dummy variables to data sets. In the next section, we'll look at some special functions related to CASE statements that handle null values specifically.

Type Conversions and Casting

Every field in a database is defined with a data type, which we reviewed at the beginning of this chapter. When data is inserted into a table, values that aren't of the field's type are rejected by the database. Strings can't be inserted into integer fields, and booleans are not allowed in date fields. Most of the time, we can take the data types for

granted and apply string functions to strings, date functions to dates, and so on. Occasionally, however, we need to override the data type of the field and force it to be something else. This is where type conversions and casting come in.

Type conversion functions allow pieces of data with the appropriate format to be changed from one data type to another. The syntax comes in a few forms that are basically equivalent. One way to change the data type is with the `cast` function, `cast (`*`input`* `as` *`data_type`*`)`, or two colons, *`input`* `::` *`data_type`*. Both of these are equivalent and convert the integer 1,234 to a string:

```
cast (1234 as varchar)

1234::varchar
```

Converting an integer to a string can be useful in CASE statements when categorizing numeric values with some unbounded upper or lower value. For example, in the following code, leaving the values that are less than or equal to 3 as integers while returning the string "4+" for higher values would result in an error:

```
case when order_items <= 3 then order_items
     else '4+'
     end
```

Casting the integers to the VARCHAR type solves the problem:

```
case when order_items <= 3 then order_items::varchar
     else '4+'
     end
```

Type conversions also come in handy when values that should be integers are parsed out of a string, and then we want to aggregate the values or use mathematical functions on them. Imagine we have a data set of prices, but the values include the dollar sign ($), and so the data type of the field is VARCHAR. We can remove the $ character with a function called `replace`, which will be discussed more during our look at text analysis in Chapter 5:

```
SELECT replace('$19.99','$','');
replace
-------
9.99
```

The result is still a VARCHAR, however, so trying to apply an aggregation will return an error. To fix this, we can `cast` the result as a FLOAT:

```
replace('$19.99','$',''):::float
cast(replace('$19.99','$','')) as float
```

Dates and datetimes can come in a bewildering array of formats, and understanding how to *cast* them to the desired format is useful. I'll show a few examples on type conversion here, and Chapter 3 will go into more detail on date and datetime calculations. As a simple example, imagine that transaction or event data often arrives in the

database as a TIMESTAMP, but we want to summarize some value such as transactions by day. Simply grouping by the timestamp will result in more rows than necessary. Casting the TIMESTAMP to a DATE reduces the size of the results and achieves our summarization goal:

```
SELECT tx_timestamp::date, count(transactions) as num_transactions
FROM ...
GROUP BY 1
;
```

Likewise, a DATE can be cast to a TIMESTAMP when a SQL function requires a TIMESTAMP argument. Sometimes the year, month, and day are stored in separate columns, or they end up as separate elements because they've been parsed out of a longer string. These then need to be assembled back into a date. To do this, we use the concatenation operator || (double pipe) or `concat` function and then cast the result to a DATE. Any of these syntaxes works and returns the same value:

```
(year || ',' || month|| '-' || day)::date
```

Or equivalently:

```
cast(concat(year, '-', month, '-', day) as date)
```

Yet another way to convert between string values and dates is by using the `date` function. For example, we can construct a string value as above and convert it into a date:

```
date(concat(year, '-', month, '-', day))
```

The *to_datatype* functions can take both a value and a format string and thus give you more control over how the data is converted. Table 2-4 summarizes the functions and their purposes. They are particularly useful when converting in and out of DATE or DATETIME formats, as they allow you to specify the order of the date and time elements.

Table 2-4. The to_datatype functions

Function	Purpose
`to_char`	Converts other types to string
`to_number`	Converts other types to numeric
`to_date`	Converts other types to date, with specified date parts
`to_timestamp`	Converts other types to date, with specified date and time parts

Sometimes the database automatically converts a data type. This is called *type coercion*. For example, INT and FLOAT numerics can usually be used together in mathematical functions or aggregations without explicitly changing the type. CHAR and VARCHAR values can usually be mixed. Some databases will coerce BOOLEAN fields to 0 and 1 values, where 0 is FALSE and 1 is TRUE, but some databases require you to convert the values explicitly. Some databases are pickier than others about

mixing dates and datetimes in result sets and functions. You can read through the documentation, or you can do some simple query experiments to learn how the database you're working with handles data types implicitly and explicitly. There is usually a way to accomplish what you want, though sometimes you need to get creative in using functions in your queries.

Dealing with Nulls: coalesce, nullif, nvl Functions

Null was one of the stranger concepts I had to get used to when I started working with data. Null just isn't something we think about in daily life, where we're used to dealing in concrete quantities of things. *Null* has a special meaning in databases and was introduced by Edgar Codd, the inventor of the relational database, to ensure that databases have a way to represent missing information. If someone asks me how many parachutes I have, I can answer "zero." But if the question is never asked, I have null parachutes.

Nulls can represent fields for which no data was collected or that aren't applicable for that row. When new columns are added to a table, the values for previously created rows will be null unless explicitly filled with some other value. When two tables are joined via an *OUTER JOIN*, nulls will appear in any fields for which there is no matching record in the second table.

Nulls are problematic for certain aggregations and groupings, and different types of databases handle them in different ways. For example, imagine I have five records, with 5, 10, 15, 20, and null. The sum of these is 50, but the average is either 10 or 12.5 depending on whether the null value is counted in the denominator. The whole question may also be considered invalid since one of the values is null. For most database functions, a null input will return a null output. Equalities and inequalities involving null also return null. A variety of unexpected and frustrating results can be output from your queries if you are not on the lookout for nulls.

When tables are defined, they can either allow nulls, reject nulls, or populate a default value if the field would otherwise be left null. In practice, this means that you can't always rely on a field to show up as null if the data is missing, because it may have been filled with a default value such as 0. I once had a long debate with a data engineer when it turned out that null dates in the source system were defaulting to "1970-01-01" in our data warehouse. I insisted that the dates should be null instead, to reflect the fact that they were unknown or not applicable. The engineer pointed out that I could remember to filter those dates or change them back to null with a CASE statement. I finally prevailed by pointing out that one day another user who wasn't as aware of the nuances of default dates would come along, run a query, and get the puzzling cluster of customers about a year before the company was even founded.

Nulls are often inconvenient or inappropriate for the analysis you want to do. They can also make output confusing to the intended audience for your analysis. Businesspeople don't necessarily understand how to interpret a null value or may assume that null values represent a problem with data quality.

Empty Strings

A concept related to but slightly different from nulls is *empty string*, where there is no value but the field is not technically null. One reason an empty string might be used is to indicate that a field is known to be blank, as opposed to null, where the value might be missing or unknown. For example, the database might have a `name_suffix` field that can be used to hold a value such as "Jr." Many people do not have a `name_suffix`, so an empty string is appropriate. Empty string can also be used as a default value instead of null, or as a way to overcome a NOT NULL constraint by inserting a value, even if empty. An empty string can be specified in a query with two quote marks:

```
WHERE my_field = '' or my_field <> 'apple'
```

Profiling the frequencies of values should reveal whether your data includes nulls, empty strings, or both.

There are a few ways to replace nulls with alternate values: CASE statements, and the specialized `coalesce` and `nullif` functions. We saw previously that CASE statements can check a condition and return a value. They can also be used to check for a null and, if one is found, replace it with another value:

```
case when num_orders is null then 0 else num_orders end

case when address is null then 'Unknown' else address end

case when column_a is null then column_b else column_a end
```

The `coalesce` function is a more compact way to achieve this. It takes two or more arguments and returns the first one that is not null:

```
coalesce(num_orders,0)

coalesce(address,'Unknown')

coalesce(column_a,column_b)

coalesce(column_a,column_b,column_c)
```

The function `nvl` exists in some databases and is similar to `coalesce`, but it allows only two arguments.

The `nullif` function compares two numbers, and if they are not equal, it returns the first number; if they *are* equal, the function returns null. Running this code:

```
nullif(6,7)
```

returns 6, whereas null is returned by:

```
nullif(6,6)
```

`nullif` is equivalent to the following, more wordy case statement:

```
case when 6 = 7 then 6
     when 6 = 6 then null
     end
```

This function can be useful for turning values back into nulls when you know a certain default value has been inserted into the database. For example, with my default time example, we could change it back to null by using:

```
nullif(date,'1970-01-01')
```

Nulls can be problematic when filtering data in the *WHERE* clause. Returning values that are null is fairly straightforward:

```
WHERE my_field is null
```

However, imagine that `my_field` contains some nulls and also some names of fruits. I would like to return all rows that are not apples. It seems like this should work:

```
WHERE my_field <> 'apple'
```

However, some databases will exclude both the "apple" rows and all rows with null values in `my_field`. To correct this, the SQL should both filter out "apple" and explicitly include nulls by connecting the conditions with OR:

```
WHERE my_field <> 'apple' or my_field is null
```

Nulls are a fact of life when working with data. Regardless of why they occur, we often need to consider them in profiling and as targets for data cleaning. Fortunately, there are a number of ways to detect them with SQL, as well as several useful functions that allow us to replace nulls with alternate values. Next we'll look at missing data, a problem that can cause nulls but has even wider implications and thus deserves a section of its own.

Missing Data

Data can be missing for a variety of reasons, each with its own implications for how you decide to handle the data's absence. A field might not have been required by the system or process that collected it, as with an optional "how did you hear about us?" field in an ecommerce checkout flow. Requiring this field might create friction for the

customer and decrease successful checkouts. Alternatively, data might normally be required but wasn't collected due to a code bug or human error, such as in a medical questionnaire where the interviewer missed the second page of questions. A change in the way the data was collected can result in records before or after the change having missing values. A tool tracking mobile app interactions might add an additional field recording whether the interaction was a tap or a scroll, for example, or remove another field due to functionality change. Data can be orphaned when a table references a value in another table, and that row or the entire table has been deleted or is not yet loaded into the data warehouse. Finally, data may be available but not at the level of detail, or granularity, needed for the analysis. An example of this comes from subscription businesses, where customers pay on an annual basis for a monthly product and we want to analyze monthly revenue.

In addition to profiling the data with histograms and frequency analysis, we can often detect missing data by comparing values in two tables. For example, we might expect that each customer in the `transactions` table also has a record in the `customer` table. To check this, query the tables using a *LEFT JOIN* and add a *WHERE* condition to find the customers that do not exist in the second table:

```
SELECT distinct a.customer_id
FROM transactions a
LEFT JOIN customers b on a.customer_id = b.customer_id
WHERE b.customer_id is null
;
```

Missing data can be an important signal in and of itself, so don't assume that it always needs to be fixed or filled. Missing data can reveal the underlying system design or biases in the data collection process.

Records with missing fields can be filtered out entirely, but often we want to keep them and instead make some adjustments based on what we know about expected or typical values. We have some options, called *imputation* techniques, for filling in missing data. These include filling with an average or median of the data set, or with the previous value. Documenting the missing data and how it was replaced is important, as this may impact the downstream interpretation and use of the data. Imputed values can be particularly problematic when the data is used in machine learning, for example.

A common option is to fill missing data with a constant value. Filling with a constant value can be useful when the value is known for some records even though they were not populated in the database. For example, imagine there was a software bug that prevented the population of the `price` for an item called "xyz," but we know the price is always $20. A CASE statement can be added to the query to handle this:

```
case when price is null and item_name = 'xyz' then 20
                                             else price
                                             end as price
```

Another option is to fill with a derived value, either a mathematical function on other columns or a CASE statement. For example, imagine we have a field for the `net_sales` amount for each transaction. Due to a bug, some rows don't have this field populated, but they do have the `gross_sales` and `discount` fields populated. We can calculate `net_sales` by subtracting `discount` from `gross_sales`:

```
SELECT gross_sales - discount as net_sales...
```

Missing values can also be filled with values from other rows in the data set. Carrying over a value from the previous row is called *fill forward*, while using a value from the next row is called *fill backward*. These can be accomplished with the `lag` and `lead` window functions, respectively. For example, imagine that our transaction table has a `product_price` field that stores the undiscounted price a customer pays for a `product`. Occasionally this field is not populated, but we can make an assumption that the price is the same as the price paid by the last customer to buy that `product`. We can fill with the previous value using the `lag` function, *PARTITION BY* the `product` to ensure the price is pulled only from the same `product`, and *ORDER BY* the appropriate date to ensure the price is pulled from the most recent prior transaction:

```
lag(product_price) over (partition by product order by order_date)
```

The `lead` function could be used to fill with `product_price` for the following transaction. Alternatively, we could take the `avg` of prices for the `product` and use that to fill in the missing value. Filling with previous, next, or average values involves making some assumptions about typical values and what's reasonable to include in an analysis. It's always a good idea to check the results to make sure they are plausible and to note that you have interpolated the data when not available.

For data that is available but not at the granularity needed, we often have to create additional rows in the data set. For example, imagine we have a `customer_subscriptions` table with the fields `subscription_date` and `annual_amount`. We can spread this annual subscription amount into 12 equal monthly revenue amounts by dividing by 12, effectively converting ARR (annual recurring revenue) into MRR (monthly recurring revenue):

```
SELECT customer_id
,subscription_date
,annual_amount
,annual_amount / 12 as month_1
,annual_amount / 12 as month_2
...
,annual_amount / 12 as month_12
FROM customer_subscriptions
;
```

This gets a bit tedious, particularly if subscription periods can be two, three, or five years as well as one year. It's also not helpful if what we want is the actual dates of the months. In theory we could write a query like this:

```
SELECT customer_id
,subscription_date
,annual_amount
,annual_amount / 12 as '2020-01'
,annual_amount / 12 as '2020-02'
...
,annual_amount / 12 as '2020-12'
FROM customer_subscriptions
;
```

However, if the data includes orders from customers across time, hardcoding the month names won't be accurate. We could use CASE statements in combination with hardcoded month names, but again this is tedious and is likely to be error-prone as you add more convoluted logic. Instead, creating new rows through a *JOIN* to a table such as a date dimension provides an elegant solution.

A *date dimension* is a static table that has one row per day, with optional extended date attributes, such as day of the week, month name, end of month, and fiscal year. The dates extend far enough into the past and far enough into the future to cover all anticipated uses. Because there are only 365 or 366 days per year, tables covering even 100 years don't take up a lot of space. Figure 2-3 shows a sample of the data in a date dimension table. Sample code to create a date dimension using SQL functions is on the book's GitHub site.

date	day_of_month	day_of_year	day_of_week	day_name	week	month_number	month_name	quarter_number	quarter_name	year	decade
2000-01-01	1	1	6	Saturday	1999-12-27	1	January	1	Q1	2000	2000
2000-01-02	2	2	0	Sunday	1999-12-27	1	January	1	Q1	2000	2000
2000-01-03	3	3	1	Monday	2000-01-03	1	January	1	Q1	2000	2000
2000-01-04	4	4	2	Tuesday	2000-01-03	1	January	1	Q1	2000	2000
2000-01-05	5	5	3	Wednesday	2000-01-03	1	January	1	Q1	2000	2000
2000-01-06	6	6	4	Thursday	2000-01-03	1	January	1	Q1	2000	2000
2000-01-07	7	7	5	Friday	2000-01-03	1	January	1	Q1	2000	2000
2000-01-08	8	8	6	Saturday	2000-01-03	1	January	1	Q1	2000	2000
2000-01-09	9	9	0	Sunday	2000-01-03	1	January	1	Q1	2000	2000
2000-01-10	10	10	1	Monday	2000-01-10	1	January	1	Q1	2000	2000
2000-01-11	11	11	2	Tuesday	2000-01-10	1	January	1	Q1	2000	2000
2000-01-12	12	12	3	Wednesday	2000-01-10	1	January	1	Q1	2000	2000
2000-01-13	13	13	4	Thursday	2000-01-10	1	January	1	Q1	2000	2000
2000-01-14	14	14	5	Friday	2000-01-10	1	January	1	Q1	2000	2000
2000-01-15	15	15	6	Saturday	2000-01-10	1	January	1	Q1	2000	2000
2000-01-16	16	16	0	Sunday	2000-01-10	1	January	1	Q1	2000	2000
2000-01-17	17	17	1	Monday	2000-01-17	1	January	1	Q1	2000	2000
2000-01-18	18	18	2	Tuesday	2000-01-17	1	January	1	Q1	2000	2000
2000-01-19	19	19	3	Wednesday	2000-01-17	1	January	1	Q1	2000	2000
2000-01-20	20	20	4	Thursday	2000-01-17	1	January	1	Q1	2000	2000
2000-01-21	21	21	5	Friday	2000-01-17	1	January	1	Q1	2000	2000
2000-01-22	22	22	6	Saturday	2000-01-17	1	January	1	Q1	2000	2000
2000-01-23	23	23	0	Sunday	2000-01-17	1	January	1	Q1	2000	2000
2000-01-24	24	24	1	Monday	2000-01-24	1	January	1	Q1	2000	2000
2000-01-25	25	25	2	Tuesday	2000-01-24	1	January	1	Q1	2000	2000
2000-01-26	26	26	3	Wednesday	2000-01-24	1	January	1	Q1	2000	2000

Figure 2-3. A date dimension table with date attributes

If you're using a Postgres database, the `generate_series` function can be used to create a date dimension either to populate the table initially or if creating a table is not an option. It takes the following form:

```
generate_series(start, stop, step interval)
```

In this function, *`start`* is the first date you want in the series, *`stop`* is the last date, and *`step interval`* is the time period between values. The *`step interval`* can take any value, but one day is appropriate for a date dimension:

```
SELECT *
FROM generate_series('2000-01-01'::timestamp,'2030-12-31', '1 day')
```

The `generate_series` function requires at least one of the arguments to be a TIMESTAMP, so "2000-01-01" is cast as a TIMESTAMP. We can then create a query that results in a row for every day, regardless of whether a customer ordered on a particular day. This is useful when we want to ensure that a customer is counted for each day, or when we specifically want to count or otherwise analyze days on which a customer did not make a purchase:

```
SELECT a.generate_series as order_date, b.customer_id, b.items
FROM
(
    SELECT *
    FROM generate_series('2020-01-01'::timestamp,'2020-12-31','1 day')
) a
LEFT JOIN
(
    SELECT customer_id, order_date, count(item_id) as items
    FROM orders
    GROUP BY 1,2
) b on a.generate_series = b.order_date
;
```

Returning to our subscription example, we can use the date dimension to create a record for each month by *JOIN*ing the date dimension on dates that are between the `subscription_date` and 11 months later (for 12 total months):

```
SELECT a.date
,b.customer_id
,b.subscription_date
,b.annual_amount / 12 as monthly_subscription
FROM date_dim a
JOIN customer_subscriptions b on a.date between b.subscription_date
and b.subscription_date + interval '11 months'
;
```

Data can be missing for various reasons, and understanding the root cause is important in deciding how to deal with it. There are a number of options for finding and replacing missing data. These include using CASE statements to set default values, deriving values by performing calculations on other fields in the same row, and interpolating from other values in the same column.

Data cleaning is an important part of the data preparation process. Data may need to be cleaned for many different reasons. Some data cleaning needs to be done to fix

poor data quality, such as when there are inconsistent or missing values in the raw data, while other data cleaning is done to make further analysis easier or more meaningful. The flexibility of SQL allows us to perform cleaning tasks in a variety of ways.

After data is cleaned, a common next step in the preparation process is shaping the data set.

Preparing: Shaping Data

Shaping data refers to manipulating the way the data is represented in columns and rows. Each table in the database has a shape. The result set of each query has a shape. Shaping data may seem like a rather abstract concept, but if you work with enough data, you will come to see its value. It is a skill that can be learned, practiced, and mastered.

One of the most important concepts in shaping data is figuring out the *granularity* of data that you need. Just as rocks can range in size from giant boulders down to grains of sand, and even further down to microscopic dust, so too can data have varying levels of detail. For example, if the population of a country is a boulder, then the population of a city is a small rock, and that of a household is a grain of sand. Data at a smaller level of detail might include individual births and deaths, or moves from one city or country to another.

Flattening data is another important concept in shaping. This refers to reducing the number of rows that represent an entity, including down to a single row. Joining multiple tables together to create a single output data set is one way to flatten data. Another way is through aggregation.

In this section, we'll first cover some considerations for choosing data shapes. Then we'll look at some common use cases: pivoting and unpivoting. We'll see examples of shaping data for specific analyses throughout the remaining chapters. Chapter 8 will go into more detail on keeping complex SQL organized when creating data sets for further analysis.

For Which Output: BI, Visualization, Statistics, ML

Deciding how to shape your data with SQL depends a lot on what you are planning to do with the data afterward. It's generally a good idea to output a data set that has as few rows as possible while still meeting your need for granularity. This will leverage the computing power of the database, reduce the time it takes to move data from the database to somewhere else, and reduce the amount of processing you or someone else needs to do in other tools. Some of the other tools that your output might go to are a BI tool for reporting and dashboarding, a spreadsheet for business users to examine, a statistics tool such as R, or a machine learning model in Python—or you might output the data straight to a visualization created with a range of tools.

When outputting data to a business intelligence tool for reports and dashboards, it's important to understand the use case. Data sets may need to be very detailed to enable exploration and slicing by end users. They may need to be small and aggregated and include specific calculations to enable fast loading and response times in executive dashboards. Understanding how the tool works, and whether it performs better with smaller data sets or is architected to perform its own aggregations across larger data sets, is important. There is no "one size fits all" answer. The more you know about how the data will be used, the better prepared you will be to shape the data appropriately.

Smaller, aggregated, and highly specific data sets often work best for visualizations, whether they are created in commercial software or using a programming language like R, Python, or JavaScript. Think about the level of aggregation and slices, or various elements, the end users will need to filter on. Sometimes the data sets require a row for each slice, as well as an "everything" slice. You may need to *UNION* together two queries—one at the detail level and one at the "everything" level.

When creating output for statistics packages or machine learning models, it's important to understand the core entity being studied, the level of aggregation desired, and the attributes or features needed. For example, a model might need one record per customer with several attributes, or a record per transaction with its associated attributes as well as customer attributes. Generally, the output for modeling will follow the notion of "tidy data" proposed by Hadley Wickham.[2] Tidy data has these properties:

1. Each variable forms a column.
2. Each observation forms a row.
3. Each value is a cell.

We will next look at how to use SQL to transform data from the structure in which it exists in your database into any other pivoted or unpivoted structure that is needed for analysis.

Pivoting with CASE Statements

A *pivot table* is a way to summarize data sets by arranging the data into rows, according to the values of an attribute, and columns, according to the values of another attribute. At the intersection of each row and column, a summary statistic such as `sum`, `count`, or `avg` is calculated. Pivot tables are often a good way to summarize data for business audiences, since they reshape the data into a more compact and easily

2 Hadley Wickham, "Tidy Data," *Journal of Statistical Software* 59, no. 10 (2014): 1–23, *https://doi.org/10.18637/jss.v059.i10*.

understandable form. Pivot tables are widely known from their implementation in Microsoft Excel, which has a drag-and-drop interface to create the summaries of data.

Pivot tables, or pivoted output, can be created in SQL using a CASE statement along with one or more aggregation functions. We've seen CASE statements several times so far, and reshaping data is another major use case for them. For example, imagine we have an `orders` table with a row for each purchase made by customers. To flatten the data, *GROUP BY* the `customer_id` and `sum` the `order_amount`:

```
SELECT customer_id
,sum(order_amount) as total_amount
FROM orders
GROUP BY 1
;

customer_id  total_amount
-----------  ------------
123          59.99
234          120.55
345          87.99
...          ...
```

To create a pivot, we will additionally create columns for each of the values of an attribute. Imagine the `orders` table also has a `product` field that contains the type of item purchased and the `order_date`. To create pivoted output, *GROUP BY* the `order_date`, and `sum` the result of a CASE statement that returns the `order_amount` whenever the row meets the product name criteria:

```
SELECT order_date
,sum(case when product = 'shirt' then order_amount
          else 0
          end) as shirts_amount
,sum(case when product = 'shoes' then order_amount
          else 0
          end) as shoes_amount
,sum(case when product = 'hat' then order_amount
          else 0
          end) hats_amount
FROM orders
GROUP BY 1
;

order_date  shirts_amount  shoes_amount  hats_amount
----------  -------------  ------------  -----------
2020-05-01  5268.56        1211.65       562.25
2020-05-02  5533.84        522.25        325.62
2020-05-03  5986.85        1088.62       858.35
...         ...            ...           ...
```

Note that with the `sum` aggregation, you can optionally use "else 0" to avoid nulls in the result set. With `count` or `count distinct`, however, you should not include an

ELSE statement, as doing so would inflate the result set. This is because the database won't count a null, but it will count a substitute value such as zero.

Pivoting with CASE statements is quite handy, and having this ability opens up data warehouse table designs that are long and narrow rather than wide, which can be better for storing sparse data, because adding columns to a table can be an expensive operation. For example, rather than storing various customer attributes in many different columns, a table could contain multiple records per customer, with each attribute in a separate row, and with `attribute_name` and `attribute_value` fields specifying what the attribute is and its value. The data can then be pivoted as needed to assemble a customer record with the desired attributes. This design is efficient when there are many sparse attributes (only a subset of customers have values for many of the attributes).

Pivoting data with a combination of aggregation and CASE statements works well when there are a finite number of items to pivot. For people who have worked with other programming languages, it's essentially looping, but written out explicitly line by line. This gives you a lot of control, such as if you want to calculate different metrics in each column, but it can also be tedious. Pivoting with case statements doesn't work well when new values arrive constantly or are rapidly changing, since the SQL code would need to be constantly updated. In those cases, pushing the computing to another layer of your analysis stack, such as a BI tool or statistical language, may be more appropriate.

Unpivoting with UNION Statements

Sometimes we have the opposite problem and need to move data stored in columns into rows instead to create tidy data. This operation is called *unpivoting*. Data sets that may need unpivoting are those that are in a pivot table format. As an example, the populations of North American countries at 10-year intervals starting in 1980 are shown in Figure 2-4.

Country	year_1980	year_1990	year_2000	year_2010
Canada	24,593	27,791	31,100	34,207
Mexico	68,347	84,634	99,775	114,061
United States	227,225	249,623	282,162	309,326

Figure 2-4. Country population by year (in thousands)[3]

3 US Census Bureau, "International Data Base (IDB)," last updated December 2020, *https://www.census.gov/data-tools/demo/idb*.

To turn this into a result set with a row per country per year, we can use a *UNION* operator. *UNION* is a way to combine data sets from multiple queries into a single result set. There are two forms, *UNION* and *UNION ALL*. When using *UNION* or *UNION ALL*, the numbers of columns in each component query must match. The data types must match or be compatible (integers and floats can be mixed, but integers and strings cannot). The column names in the result set come from the first query. Aliasing the fields in the remaining queries is therefore optional but can make a query easier to read:

```
SELECT country
,'1980' as year
,year_1980 as population
FROM country_populations
    UNION ALL
SELECT country
,'1990' as year
,year_1990 as population
FROM country_populations
    UNION ALL
SELECT country
,'2000' as year
,year_2000 as population
FROM country_populations
    UNION ALL
SELECT country
,'2010' as year
,year_2010 as population
FROM country_populations
;

country        year  population
-------------  ----  ----------
Canada         1980  24593
Mexico         1980  68347
United States  1980  227225
...            ...   ...
```

In this example, we use a constant to hardcode the year, in order to keep track of the year that the population value corresponds to. The hardcoded values can be of any type, depending on your use case. You may need to explicitly cast certain hardcoded values, such as when entering a date:

```
'2020-01-01'::date as date_of_interest
```

What is the difference between *UNION* and *UNION ALL*? Both can be used to append or stack data together in this fashion, but they are slightly different. *UNION* removes duplicates from the result set, whereas *UNION ALL* retains all records, whether duplicates or not. *UNION ALL* is faster, since the database doesn't have to do a pass over the data to find duplicates. It also ensures that every record ends up in the result set. I tend to use *UNION ALL*, using *UNION* only when I have a reason to suspect duplicate data.

*UNION*ing data can also be useful for bringing together data from different sources. For example, imagine we have a `populations` table with yearly data per country, and another `gdp` table with yearly gross domestic product, or GDP. One option is to *JOIN* the tables and obtain a result set with one column for population and another for GDP:

```
SELECT a.country, a.population, b.gdp
FROM populations a
JOIN gdp b on a.country = b.country
;
```

Another option is to *UNION ALL* the data sets so that we end up with a stacked data set:

```
SELECT country, 'population' as metric, population as metric_value
FROM populations
    UNION ALL
SELECT country, 'gdp' as metric, gdp as metric_value
FROM gdp
;
```

Which approach you use largely depends on the output that you need for your analysis. The latter option can be useful when you have a number of different metrics in different tables and no single table has a full set of entities (in this case, countries). This is an alternative approach to a *FULL OUTER JOIN*.

pivot and unpivot Functions

Recognizing that the pivot and unpivot use cases are common, some database vendors have implemented functions to do this with fewer lines of code. Microsoft SQL Server and Snowflake have `pivot` functions that take the form of extra expressions in the *WHERE* clause. Here, aggregation is any aggregation function, such as `sum` or `avg`, the `value_column` is the field to be aggregated, and a column will be created for each value of the `label_column` listed as a label:

```
SELECT...
FROM...
    pivot(aggregation(value_column)
          for label_column in (label_1, label_2, ...)
;
```

We could rewrite the earlier pivoting example that used CASE statements as follows:

```
SELECT *
FROM orders
  pivot(sum(order_amount) for product in ('shirt','shoes'))
GROUP BY order_date
;
```

Although this syntax is more compact than the CASE construction we saw earlier, the desired columns still need to be specified. As a result, `pivot` doesn't solve the problem of newly arriving or rapidly changing sets of fields that need to be turned into columns. Postgres has a similar `crosstab` function, available in the `tablefunc` module.

Microsoft SQL Server and Snowflake also have `unpivot` functions that work in a similar fashion to expressions in the *WHERE* clause and transform rows into columns:

```
SELECT...
FROM...
    unpivot( value_column for label_column in (label_1, label_2, ...))
;
```

For example, the `country_populations` data from the previous example could be reshaped in the following manner:

```
SELECT *
FROM country_populations
    unpivot(population for year in (year_1980, year_1990, year_2000, year_2010))
;
```

Here again the syntax is more compact than the *UNION* or *UNION ALL* approach we looked at earlier, but the list of columns must be specified in the query.

Postgres has an `unnest` array function that can be used to unpivot data, thanks to its array data type. An array is a collection of elements, and in Postgres you can list the elements of an array in square brackets. The function can be used in the *SELECT* clause and takes this form:

```
unnest(array[element_1, element_2, ...])
```

Returning to our earlier example with countries and populations, this query returns the same result as the query with the repeated *UNION ALL* clauses:

```
SELECT
country
,unnest(array['1980', '1990', '2000', '2010']) as year
,unnest(array[year_1980, year_1990, year_2000, year_2010]) as pop
FROM country_populations
;
```

```
country  year  pop
-------  ----  -----
Canada   1980  24593
Canada   1990  27791
Canada   2000  31100
...      ...   ...
```

Data sets arrive in many different formats and shapes, and they aren't always in the format needed in our output. There are several options for reshaping data through pivoting or unpivoting it, either with CASE statements or *UNION*s, or with database-specific functions. Understanding how to manipulate your data in order to shape it in the way you want will give you greater flexibility in your analysis and in the way you present your results.

Conclusion

Preparing data for analysis can feel like the work you do before you get to the real work of analysis, but it is so fundamental to understanding the data that I always find it is time well spent. Understanding the different types of data you're likely to encounter is critical, and you should take the time to understand the data types in each table you work with. Profiling data helps us learn more about what is in the data set and examine it for quality. I often return to profiling throughout my analysis projects, as I learn more about the data and need to check my query results along the way as I build in complexity. Data quality will likely never stop being a problem, so we've looked at some ways to handle the cleaning and enhancement of data sets. Finally, knowing how to shape the data to create the right output format is essential. We'll see these topics recur in the context of various analyses throughout the book. The next chapter, on time series analysis, starts our journey into specific analysis techniques.

CHAPTER 3
Time Series Analysis

Now that I've covered SQL and databases and the key steps in preparing data for analysis, it's time to turn to specific types of analysis that can be done with SQL. There are a seemingly unending number of data sets in the world, and correspondingly infinite ways in which they could be analyzed. In this and the following chapters, I have organized types of analysis into themes that I hope will be helpful as you build your analysis and SQL skills. Many of the techniques to be discussed build on those shown in Chapter 2 and then on the preceding chapters as the book progresses. Time series of data are so prevalent and so important that I'll start the series of analysis themes here.

Time series analysis is one of the most common types of analysis done with SQL. A *time series* is a sequence of measurements or data points recorded in time order, often at regularly spaced intervals. There are many examples of time series data in daily life, such as the daily high temperature, the closing value of the S&P 500 stock index, or the number of daily steps recorded by your fitness tracker. Time series analysis is used in a wide variety of industries and disciplines, from statistics and engineering to weather forecasting and business planning. Time series analysis is a way to understand and quantify how things change over time.

Forecasting is a common goal of time series analysis. Since time only marches forward, future values can be expressed as a function of past values, while the reverse is not true. However, it's important to note that the past doesn't perfectly predict the future. Any number of changes to wider market conditions, popular trends, product introductions, or other large changes make forecasting difficult. Still, looking at historical data can lead to insights, and developing a range of plausible outcomes is useful for planning. As I'm writing this, the world is in the midst of a global COVID-19 pandemic, the likes of which haven't been seen in 100 years—predating all but the most long-lived organizations' histories. Thus many current organizations haven't

seen this specific event before, but they have existed through other economic crises, such as those following the dot-com burst and the 9/11 attacks in 2001, as well as the global financial crisis of 2007–2008. With careful analysis and understanding of context, we can often extract useful insights.

In this chapter, we'll first cover the SQL building blocks of time series analysis: syntax and functions for working with dates, timestamps, and time. Next, I'll introduce the retail sales data set used for examples throughout the rest of the chapter. A discussion of methods for trending analysis follows, and then I'll cover calculating rolling time windows. Next are period-over-period calculations to analyze data with seasonality components. Finally, we'll wrap up with some additional techniques that are useful for time series analysis.

Date, Datetime, and Time Manipulations

Dates and times come in a wide variety of formats, depending on the data source. We often need or want to transform the raw data format for our output, or to perform calculations to arrive at new dates or parts of dates. For example, the data set might contain transaction timestamps, but the goal of the analysis is to trend monthly sales. At other times, we might want to know how many days or months have elapsed since a particular event. Fortunately, SQL has powerful functions and formatting capabilities that can transform just about any raw input to almost any output we might need for analysis.

In this section, I'll show you how to convert between time zones, and then I'll go into depth on formatting dates and datetimes. Next, I'll explore date math and time manipulations, including those that make use of intervals. An interval is a data type that holds a span of time, such as a number of months, days, or hours. Although data can be stored in a database table as an interval type, in practice I rarely see this done, so I will talk about intervals alongside the date and time functions that you can use them with. Last, I'll discuss some special considerations when joining or otherwise combining data from different sources.

Time Zone Conversions

Understanding the standard time zone used in a data set can prevent misunderstandings and mistakes further into the analysis process. Time zones split the world into north-south regions that observe the same time. Time zones allow different parts of the world to have similar clock times for daytime and nighttime—so, for example, the sun is overhead at 12 p.m. wherever you are in the world. The zones follow irregular boundaries that are as much political as geographic ones. Most are one hour apart, but some are offset only 30 or 45 minutes, and so there are more than 30 time zones spanning the globe. Many countries that are distant from the equator observe daylight savings time for parts of the year as well, but there are exceptions, such as in the

United States and Australia, where some states observe daylight savings time and others do not. Each time zone has a standard abbreviation, such as PST for Pacific Standard Time and PDT for Pacific Daylight Time.

Many databases are set to *Coordinated Universal Time* (UTC), the global standard used to regulate clocks, and record events in this time zone. It replaced *Greenwich Mean Time* (GMT), which you might still see if your data comes from an older database. UTC does not have daylight savings time, so it stays consistent all year long. This turns out to be quite useful for analysis. I remember one time a panicked product manager asked me to figure out why sales on a particular Sunday dropped so much compared to the prior Sunday. I spent hours writing queries and investigating possible causes before eventually figuring out that our data was recorded in Pacific Time (PT). Daylight savings started early Sunday morning, the database clock moved ahead 1 hour, and the day had only 23 hours instead of 24, and thus sales appeared to drop. Half a year later we had a corresponding 25-hour day, when sales appeared unusually high.

Often timestamps in the database are not encoded with the time zone, and you will need to consult with the source or developer to figure out how your data was stored. UTC has become most common in the data sets I see, but that is certainly not universal.

One drawback to UTC, or really to any logging of machine time, is that we lose information about the local time for the human doing the actions that generated the event recorded in the database. I might want to know whether people tend to use my mobile app more during the workday or during nights and weekends. If my audience is clustered in one time zone, it's not hard to figure this out. But if the audience spans multiple time zones or is international, then it becomes a calculation task of converting each recorded time to its local time zone.

All local time zones have a UTC offset. For example, the offset for PDT is UTC – 7 hours, while the offset for PST is UTC – 8 hours. Timestamps in databases are stored in the format YYYY-MM-DD hh:mi:ss (for years-months-days hours:minutes:seconds). Timestamps with the time zone have an additional piece of information for the UTC offset, expressed as a positive or negative number. Converting from one time zone to another can be accomplished with `at time zone` followed by the destination time zone's abbreviation. For example, we can convert a timestamp in UTC (offset – 0) to PST:

```
SELECT '2020-09-01 00:00:00 -0' at time zone 'pst';

timezone
-------------------
2020-08-31 16:00:00
```

The destination time zone name can be a constant, or a database field, allowing this conversion to be dynamic to the data set. Some databases have a `convert_timezone` or `convert_tz` function that works similarly. One argument is the time zone of the result, and the other argument is the time zone from which to convert:

```
SELECT convert_timezone('pst','2020-09-01 00:00:00 -0');

timezone
-------------------
2020-08-31 16:00:00
```

Check your database's documentation for the exact name and ordering of the target time zone and the source timestamp arguments. Many databases contain a list of time zones and their abbreviations in a system table. Some common ones are seen in Table 3-1. These can be queried with *SELECT * FROM* the table name. Wikipedia also has a useful list of standard time zone abbreviations and their UTC offsets.

Table 3-1. Time zone information system tables in common databases

Postgres	`pg_timezone_names`
MySQL	`mysql.time_zone_names`
SQL Server	`sys.time_zone_info`
Redshift	`pg_timezone_names`

Time zones are an innate part of working with timestamps. With time zone conversion functions, moving between the time zone in which the data was recorded and any other world time zone is possible. Next, I'll show you a variety of techniques for manipulating dates and timestamps with SQL.

Date and Timestamp Format Conversions

Dates and timestamps are key to time series analysis. Due to the wide variety of ways in which dates and times can be represented in source data, it is almost inevitable that you will need to convert date formats at some point. In this section, I'll cover several of the most common conversions and how to accomplish them with SQL: changing the data type, extracting parts of a date or timestamp, and creating a date or timestamp from parts. I'll begin by introducing some handy functions that return the current date and/or time.

Returning the current date or time is a common analysis task—for example, to include a timestamp for the result set or to use in date math, covered in the next section. The current date and time are referred to as *system time*, and while returning them is easy to do with SQL, there are some syntax differences between databases.

To return the current date, some databases have a `current_date` function, with no parentheses:

```
SELECT current_date;
```

There is a wider variety of functions to return the current date and time. Check your database's documentation or just experiment by typing into a SQL window to see whether a function returns a value or an error. The functions with parentheses do not take arguments, but it is important to include the parentheses:

```
current_timestamp
localtimestamp
get_date()
now()
```

Finally, there are functions to return only the timestamp portion of the current system time. Again, consult documentation or experiment to figure out which function(s) to use with your database:

```
current_time
localtime
timeofday()
```

SQL has a number of functions for changing the format of dates and times. To reduce the granularity of a timestamp, use the `date_trunc` function. The first argument is a text value indicating the time period level to which to truncate the timestamp in the second argument. The result is a timestamp value:

```
date_trunc (text, timestamp)

SELECT date_trunc('month','2020-10-04 12:33:35'::timestamp);

date_trunc
-------------------
2020-10-01 00:00:00
```

Standard arguments that can be used are listed in Table 3-2. They range all the way from microseconds to millennia, providing plenty of flexibility. Databases that don't support `date_trunc`, such as MySQL, have an alternate function called `date_format` that can be used in a similar way:

```
SELECT date_format('2020-10-04 12:33:35','%Y-%m-01') as date_trunc;

date_trunc
-------------------
2020-10-01 00:00:00
```

Table 3-2. Standard time period arguments

Time period arguments
microsecond
millisecond
second
minute
hour
day
week
month
quarter
year
decade
century
millennium

Rather than returning dates or timestamps, sometimes our analysis calls for parts of dates or times. For example, we might want to group sales by month, day of the week, or hour of the day.

SQL provides a few functions for returning just the part of the date or timestamp required. Dates and timestamps are usually interchangeable, except when the request is to return a time part. In those cases, time is of course required.

The `date_part` function takes a text value for the part to be returned and a date or timestamp value. The returned value is a FLOAT, which is a numeric value with a decimal part; depending on your needs, you may want to cast the value to an integer data type:

```
SELECT date_part('day',current_timestamp);
SELECT date_part('month',current_timestamp);
SELECT date_part('hour',current_timestamp);
```

Another function that works similarly is `extract`, which takes a part name and a date or timestamp value and returns a FLOAT value:

```
SELECT extract('day' from current_timestamp);

date_part
---------
27.0

SELECT extract('month' from current_timestamp);
```

```
date_part
---------
5.0

SELECT extract('hour' from current_timestamp);

date_part
---------
14.0
```

The functions `date_part` and `extract` can be used with intervals, but note that the requested part must match the units of the interval. So, for example, requesting days from an interval stated in days returns the expected value of 30:

```
SELECT date_part('day',interval '30 days');

SELECT extract('day' from interval '30 days');

date_part
---------
30.0
```

However, requesting days from an interval stated in months returns a value of 0.0:

```
SELECT extract('day' from interval '3 months');

date_part
---------
0.0
```

A full list of date parts can be found in your database's documentation or by searching online, but some of the most common are "day," "month," and "year" for dates, and "second," "minute," and "hour" for timestamps.

To return text values of the date parts, use the `to_char` function, which takes the input value and the output format as arguments:

```
SELECT to_char(current_timestamp,'Day');
SELECT to_char(current_timestamp,'Month');
```

If you ever encounter timestamps stored as Unix epochs (the number of seconds that have elapsed since January 1, 1970, at 00:00:00 UTC), you can convert them to timestamps using the `to_timestamp` function.

Sometimes analysis calls for creating a date from parts from different sources. This can occur when the year, month, and day values are stored in different columns in the

database. It can also be necessary when the parts have been parsed out of text, a topic I'll cover in more depth in Chapter 5.

A simple way to create a timestamp from separate date and time components is to concatenate them together with a plus sign (+):

```
SELECT date '2020-09-01' + time '03:00:00' as timestamp;

timestamp
-------------------
2020-09-01 03:00:00
```

A date can be assembled using the `make_date`, `makedate`, `date_from_parts`, or `date fromparts` function. These are equivalent, but different databases name the functions differently. The function takes arguments for the year, month, and day parts and returns a value with a date format:

```
SELECT make_date(2020,09,01);

make_date
----------
2020-09-01
```

The arguments can be constants or reference field names and must be integers. Yet another way to assemble a date or timestamp is to concatenate the values together and then cast the result to a date format using one of the casting syntaxes or the `to_date` function:

```
SELECT to_date(concat(2020,'-',09,'-',01), 'yyyy-mm-dd');

to_date
----------
2020-09-01

SELECT cast(concat(2020,'-',09,'-',01) as date);

to_date
----------
2020-09-01
```

SQL has a number of ways to format and convert dates and timestamps and retrieve system dates and times. In the next section, I will start putting them to use in date math.

Date Math

SQL allows us to do various mathematical operations on dates. This might be surprising since, strictly speaking, dates are not numeric data types, but the concept should be familiar if you've ever tried to figure out what day it will be four weeks from now. Date math is useful for a variety of analytics tasks. For example, we can use it to find

the age or tenure of a customer, how much time elapsed between two events, and how many things occurred within a window of time.

Date math involves two types of data: the dates themselves and intervals. We need the concept of intervals because date and time components don't behave like integers. One-tenth of 100 is 10; one-tenth of a year is 36.5 days. Half of 100 is 50; half of a day is 12 hours. Intervals allow us to move smoothly between units of time. Intervals come in two types: year-month intervals and day-time ones. We'll start with a few operations that return integer values and then move on to functions that work with or return intervals.

First, let's find the days elapsed between two dates. There are several ways to do this in SQL. The first way is by using a mathematical operator, the minus sign (–):

```
SELECT date('2020-06-30') - date('2020-05-31') as days;

days
----
30
```

This returns the number of days between these two dates. Note that the answer is 30 days and not 31. The number of days is inclusive of only one of the endpoints. Subtracting the dates in the reverse also works and returns an interval of –30 days:

```
SELECT date('2020-05-31') - date('2020-06-30') as days;

days
----
-30
```

Finding the difference between two dates can also be accomplished with the `datediff` function. Postgres does not support it, but many other popular databases do, including SQL Server, Redshift, and Snowflake, and it's quite handy, particularly when the goal is to return an interval other than the number of days. The function takes three arguments—the time period units you want to return, a starting timestamp or date, and an ending timestamp or date:

```
datediff(interval_name, start_timestamp, end_timestamp)
```

So our previous example would look like this:

```
SELECT datediff('day',date('2020-05-31'), date('2020-06-30')) as days;

days
----
30
```

We can also find the number of months between two dates, and the database will do the correct math even though month lengths differ throughout the year:

```
SELECT datediff('month'
                ,date('2020-01-01')
                ,date('2020-06-30')
                ) as months;

months
------
5
```

In Postgres, this can be accomplished using the `age` function, which calculates the interval between two dates:

```
SELECT age(date('2020-06-30'),date('2020-01-01'));

age
--------------
5 mons 29 days
```

We can then find the number of months component of the interval with the `date_part()` function:

```
SELECT date_part('month',age('2020-06-30','2020-01-01')) as months;

months
------
5.0
```

Subtracting dates to find the time elapsed between them is quite powerful. Adding dates does not work in the same way. To do addition with dates, we need to leverage intervals or special functions.

For example, we can add seven days to a date by adding the interval `'7 days'`:

```
SELECT date('2020-06-01') + interval '7 days' as new_date;

new_date
-------------------
2020-06-08 00:00:00
```

Some databases don't require the interval syntax and instead automatically convert the provided number to days, although it's generally good practice to use the interval notation, both for cross-database compatibility and to make your code easier to read:

```
SELECT date('2020-06-01') + 7 as new_date;

new_date
-------------------
2020-06-08 00:00:00
```

If you want to add a different unit of time, use the interval notation with months, years, hours, or another date or time period. Note that this can also be used to subtract intervals from dates by using a "-" instead of a "+." Many but not all databases

have a `date_add` or `dateadd` function that takes the desired interval, a value, and the starting date and does the math:

```
SELECT date_add('month',1,'2020-06-01') as new_date;

new_date
----------
2020-07-01
```

Consult your database's documentation, or just experiment with queries, to figure out the syntax and functions that are available and appropriate for your project.

Any of these formulations can be used in the *WHERE* clause in addition to the *SELECT* clause. For example, we can filter to records that occurred at least three months ago:

```
WHERE event_date < current_date - interval '3 months'
```

They can also be used in *JOIN* conditions, but note that database performance will usually be slower when the *JOIN* condition contains a calculation rather than an equality or inequality between dates.

Using date math is common in analysis with SQL, both to find the time elapsed between dates or timestamps and to calculate new dates based on an interval from a known date. There are several ways to find the elapsed time between two dates, add intervals to dates, and subtract intervals from dates. Next, we'll turn to time manipulations, which are similar.

Time Math

Time math is less common in many areas of analysis, but it can be useful in some situations. For example, we might want to know how long it takes for a support representative to answer a phone call in a call center or respond to an email requesting assistance. Whenever the elapsed time between two events is less than a day, or when rounding the result to a number of days doesn't provide enough information, time manipulation comes into play. Time math works similarly to date math, by leveraging intervals. We can add time intervals to times:

```
SELECT time '05:00' + interval '3 hours' as new_time;

new_time
--------
08:00:00
```

We can subtract intervals from times:

```
SELECT time '05:00' - interval '3 hours' as new_time;

new_time
--------
02:00:00
```

We can also subtract times, resulting in an interval:

```
SELECT time '05:00' - time '03:00' as time_diff;

time_diff
---------
02:00:00
```

Times, unlike dates, can be multiplied:

```
SELECT time '05:00' * 2 as time_multiplied;

time_multiplied
---------------
10:00:00
```

Intervals can also be multiplied, resulting in a time value:

```
SELECT interval '1 second' * 2000 as interval_multiplied;

interval_multiplied
-------------------
00:33:20


SELECT interval '1 day' * 45 as interval_multiplied;

interval_multiplied
-------------------
45 days
```

These examples use constant values, but you can include database field names or calculations in the SQL query as well to make the calculations dynamic. Next, I'll discuss special date considerations to keep in mind when combining data sets from different source systems.

Joining Data from Different Sources

Combining data from different sources is one of the most compelling use cases for a data warehouse. However, different source systems can record dates and times in different formats or different time zones or even just be off slightly due to issues with the internal clock time of the server. Even tables from the same data source can have differences, though this is less common. Reconciling and standardizing dates and timestamps is an important step before moving further in the analysis.

Dates and timestamps that are in different formats can be standardized with SQL. *JOIN*ing on dates or including date fields in *UNION*s generally requires that the dates or timestamps be in the same format. Earlier in the chapter, I showed techniques for formatting dates and timestamps that will serve well with these problems. Take care with time zones when combining data from different sources. For example, an internal database may use UTC time, but data from a third party could be in a local time zone. I have seen data sourced from software as a service (SaaS) that was recorded in a variety of local times. Note that the timestamp values themselves won't necessarily have the time zone embedded. You may need to consult the vendor's documentation and convert the data to UTC if the rest of your data is stored that way. Another option is to store the time zone in a field so that the timestamp value can be converted as needed.

Another thing to look out for when working with data from different sources is timestamps that are slightly out of sync. This can happen when timestamps are recorded from client devices—for example, from a laptop or mobile phone in one data source and a server in the other data source. I once saw a series of experiment results be miscalculated because the client mobile device that recorded a user's action was offset by a few minutes from the server that recorded the treatment group to which the user was assigned. Data from the mobile clients appeared to arrive before the treatment group timestamp, so some events were inadvertently excluded. A fix for something like this is relatively straightforward: rather than filter for action timestamps greater than the treatment group timestamp, allow events within a short interval or window of time prior to the treatment timestamp to be included in the results. This can be accomplished with a *BETWEEN* clause and date math, as seen in the last section.

When working with data from mobile apps, pay particular attention to whether the timestamps represent when the action happened on the device *or* when the event arrived in the database. The difference can range from negligible all the way up to days, depending on whether the mobile app allows offline usage and on how it handles sending data during periods of low signal strength. Data from mobile apps can be late-arriving or may make its way into the database days after it occurred on the device. Dates and timestamps can also become corrupted en route, and you may see ones that are impossibly distant in the past or future as a result.

Now that I've shown how to manipulate dates, datetimes, and time by changing the formats, converting time zones, performing date math, and working across data sets from different sources, we're ready to get into some time series examples. First, I'll introduce the data set for examples in the rest of the chapter.

The Retail Sales Data Set

The examples in the rest of this chapter use a data set of monthly US retail sales from the Monthly Retail Trade Report: Retail and Food Services Sales: Excel (1992–present), available on the Census.gov website. The data in this report is used as an economic indicator to understand trends in US consumer spending patterns. While gross domestic product (GDP) figures are published quarterly, this retail sales data is published monthly, so it is also used to help predict GDP. For both of these reasons, the latest figures are usually covered in the business press when they are released.

The data spans from 1992 to 2020 and includes both total sales as well as details for subcategories of retail sales. It contains both unadjusted and seasonally adjusted numbers. This chapter will use the unadjusted numbers, since one of the goals is analyzing seasonality. Sales figures are in millions of US dollars. The original file format is an Excel file, with a tab for each year and with months as columns. The GitHub site for this book has the data in a format that's easier to import into a database, along with code specifically for importing into Postgres. Figure 3-1 shows a sample of the `retail_sales` table.

*	sales_month	naics_code	kind_of_business	reason_for_null	sales
1	2020-01-01	441	Motor vehicle and parts dealers	(null)	93268
2	2020-01-01	4411	Automobile dealers	(null)	80728
3	2020-01-01	4411, 4412	Automobile and other motor vehicle dealers	(null)	85823
4	2020-01-01	44111	New car dealers	(null)	71757
5	2020-01-01	44112	Used car dealers	(null)	8971
6	2020-01-01	4413	Automotive parts, acc., and tire stores	(null)	7445
7	2020-01-01	442	Furniture and home furnishings stores	(null)	9257
8	2020-01-01	442, 443	Furniture, home furn, electronics, and appliance stores	(null)	16993
9	2020-01-01	4421	Furniture stores	(null)	4904
10	2020-01-01	4422	Home furnishings stores	(null)	4353
11	2020-01-01	44221	Floor covering stores	Supressed	(null)
12	2020-01-01	442299	All other home furnishings stores	(null)	2408
13	2020-01-01	443	Electronics and appliance stores	(null)	7736
14	2020-01-01	443141	Household appliance stores	(null)	1197
15	2020-01-01	443142	Electronics stores	(null)	6539
16	2020-01-01	444	Building mat. and garden equip. and supplies dealers	(null)	27887
17	2020-01-01	4441	Building mat. and supplies dealers	(null)	24555
18	2020-01-01	44412	Paint and wallpaper stores	(null)	903
19	2020-01-01	44413	Hardware stores	(null)	1902
20	2020-01-01	445	Food and beverage stores	(null)	63590
21	2020-01-01	4451	Grocery stores	(null)	57667
22	2020-01-01	44511	Supermarkets and other grocery (except convenience) stores	(null)	55178
23	2020-01-01	4453	Beer, wine, and liquor stores	(null)	4388
24	2020-01-01	446	Health and personal care stores	(null)	30047
25	2020-01-01	44611	Pharmacies and drug stores	(null)	25209

Figure 3-1. Preview of the US retail sales data set

Trending the Data

With time series data, we often want to look for trends in the data. A trend is simply the direction in which the data is moving. It may be moving up or increasing over time, or it may be moving down or decreasing over time. It can remain more or less flat, or there could be so much noise, or movement up and down, that it's hard to determine a trend at all. This section will cover several techniques for trending time series data, from simple trends for graphing to comparing components of a trend, using percent of total calculations to compare parts to the whole, and finally indexing to see the percent change from a reference time period.

Simple Trends

Creating a trend may be a step in profiling and understanding data, or it may be the final output. The result set is a series of dates or timestamps and a numerical value. When graphing a time series, the dates or timestamps will become the x-axis, and the numerical value will be the y-axis. For example, we can check the trend of total retail and food services sales in the US:

```
SELECT sales_month
,sales
FROM retail_sales
WHERE kind_of_business = 'Retail and food services sales, total'
;

sales_month  sales
-----------  ------
1992-01-01   146376
1992-02-01   147079
1992-03-01   159336
...          ...
```

The results are graphed in Figure 3-2.

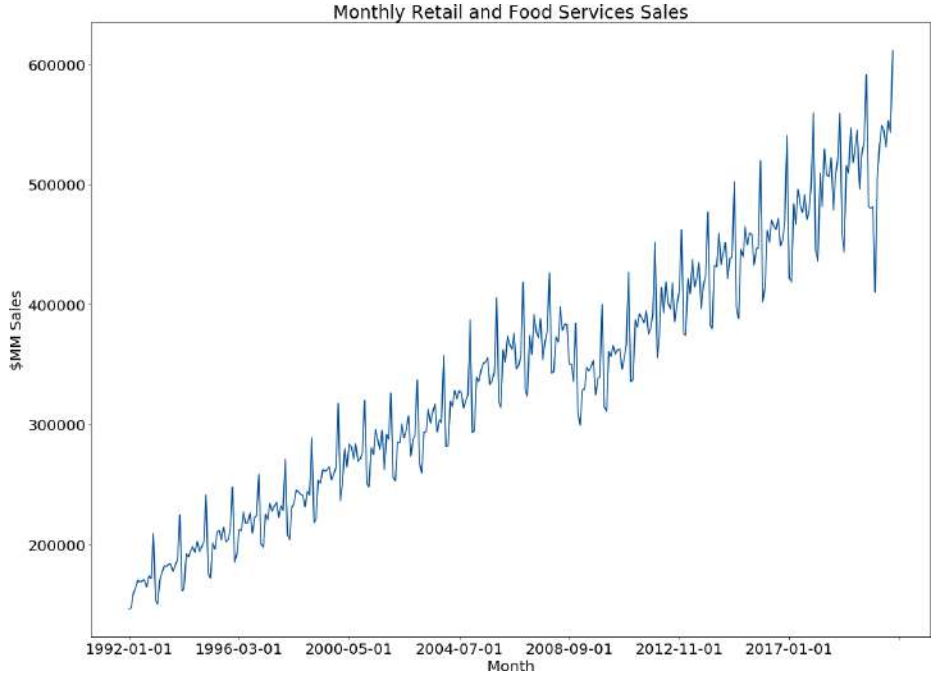


Figure 3-2. Trend of monthly retail and food services sales

This data clearly has some patterns, but it also has some noise. Transforming the data and aggregating at the yearly level can help us gain a better understanding. First, we'll use the `date_part` function to return just the year from the `sales_month` field and then `sum` the `sales`. The results are filtered to the "Retail and food services sales, total" `kind_of_business` in the *WHERE* clause:

```
SELECT date_part('year',sales_month) as sales_year
,sum(sales) as sales
FROM retail_sales
WHERE kind_of_business = 'Retail and food services sales, total'
GROUP BY 1
;

sales_year  sales
----------  -------
1992.0      2014102
1993.0      2153095
1994.0      2330235
...         ...
```

After graphing this data, as in Figure 3-3, we now have a smoother time series that is generally increasing over time, as might be expected, since the sales values are not adjusted for inflation. Sales for all retail and food services fell in 2009, during the

global financial crisis. After growing every year throughout the 2010s, sales were flat in 2020 compared to 2019, due to the impact of the COVID-19 pandemic.

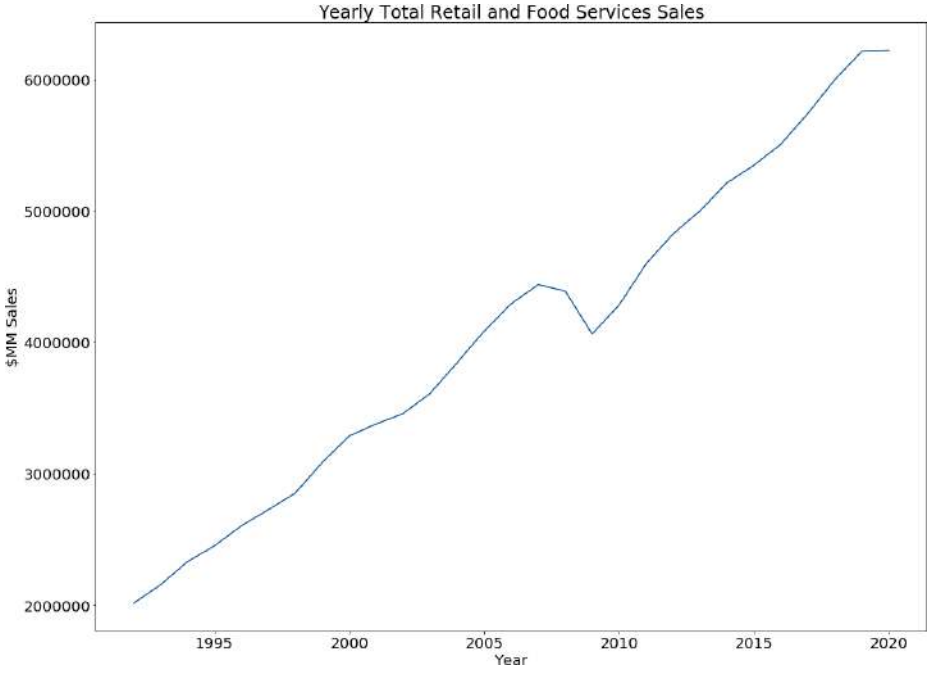


Figure 3-3. Trend of yearly total retail and food services sales

Graphing time series data at different levels of aggregation, such as weekly, monthly, or yearly, is a good way to understand trends. This step can be used to simply profile the data, but it can also be the final output, depending on the goals of the analysis. Next, we'll turn to using SQL to compare components of a time series.

Comparing Components

Often data sets contain not just a single time series but multiple slices or components of a total across the same time range. Comparing these slices often reveals interesting patterns. In the retail sales data set, there are values for total sales but also a number of subcategories. Let's compare the yearly sales trend for a few categories that are associated with leisure activities: book stores, sporting goods stores, and hobby stores. This query adds `kind_of_business` in the *SELECT* clause and, since it is another attribute rather than an aggregation, adds it to the *GROUP BY* clause as well:

```
SELECT date_part('year',sales_month) as sales_year
,kind_of_business
,sum(sales) as sales
FROM retail_sales
```

```
WHERE kind_of_business in ('Book stores'
 ,'Sporting goods stores','Hobby, toy, and game stores')
GROUP BY 1,2
;

sales_year  kind_of_business             sales
----------  --------------------------  -----
1992.0      Book stores                 8327
1992.0      Hobby, toy, and game stores 11251
1992.0      Sporting goods stores       15583
...         ...                         ...
```

The results are graphed in Figure 3-4. Sales at sporting goods retailers started the highest among the three categories and grew much faster during the time period, and by the end of the time series, those sales were substantially higher. Sales at sporting goods stores started declining in 2017 but had a big rebound in 2020. Sales at hobby, toy, and game stores were relatively flat over this time span, with a slight dip in the mid-2000s and another slight decline prior to a rebound in 2020. Sales at book stores grew until the mid-2000s and have been on the decline since then. All of these categories have been impacted by the growth of online retailers, but the timing and magnitude seem to differ.

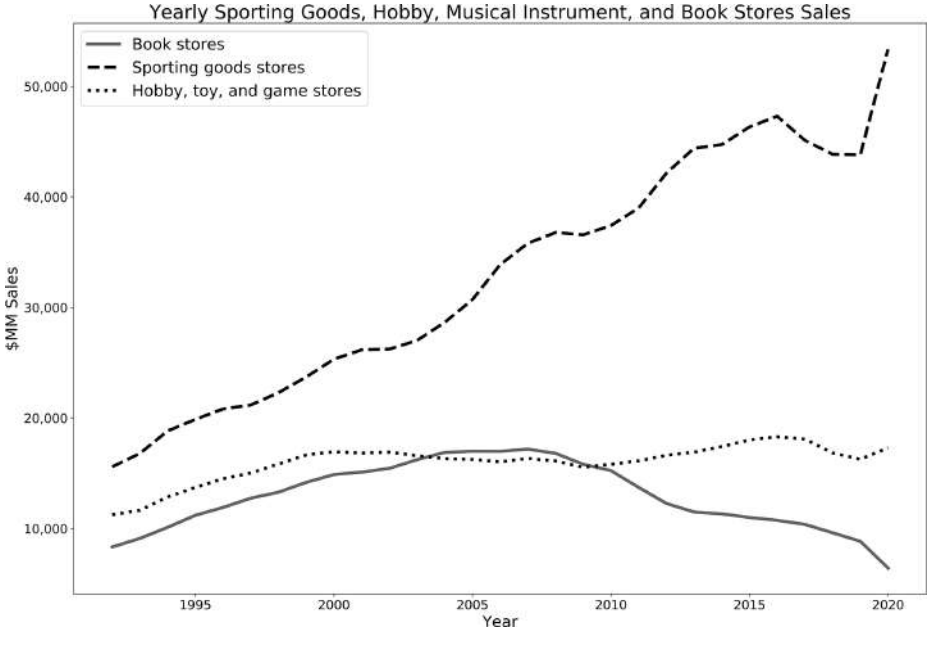


Figure 3-4. Trend of yearly retail sales for sporting goods stores; hobby, toy, and game stores; and book stores

In addition to looking at simple trends, we might want to perform more complex comparisons between parts of the time series. For the next few examples, we'll look at sales at women's clothing stores and at men's clothing stores. Note that since the names contain apostrophes, the character otherwise used to indicate the beginning and end of strings, we need to escape them with an extra apostrophe. This lets the database know that the apostrophe is part of the string rather than the end. Although we might consider adding a step in a data-loading pipeline that removes extra apostrophes in names, I've left them in here as a demonstration of the types of code adjustments that are often needed in the real world. First, we'll trend the data for each type of store by month:

```
SELECT sales_month
,kind_of_business
,sales
FROM retail_sales
WHERE kind_of_business in ('Men''s clothing stores'
 ,'Women''s clothing stores')
;

sales_month  kind_of_business         sales
-----------  ----------------------  -----
1992-01-01   Men's clothing stores    701
1992-01-01   Women's clothing stores  1873
1992-02-01   Women's clothing stores  1991
...          ...                     ...
```

The results are graphed in Figure 3-5. Sales at women's clothing retailers are much higher than those at men's clothing retailers. Both types of stores exhibit seasonality, a topic I'll cover in depth in "Analyzing with Seasonality" on page 107. Both experienced significant drops in 2020 due to store closures and a reduction in shopping because of the COVID-19 pandemic.

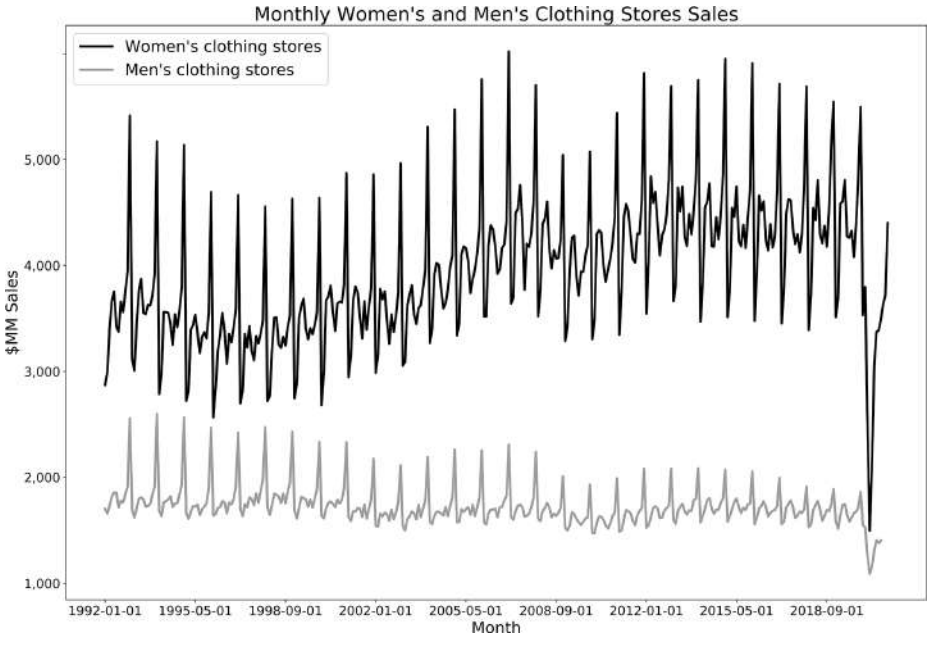


Figure 3-5. Monthly trend of sales at women's and men's clothing stores

The monthly data has intriguing patterns but is noisy, so we'll use yearly aggregates for the next few examples. We've seen this query format previously when rolling up total sales and sales for leisure categories:

```
SELECT date_part('year',sales_month) as sales_year
,kind_of_business
,sum(sales) as sales
FROM retail_sales
WHERE kind_of_business in ('Men''s clothing stores'
 ,'Women''s clothing stores')
GROUP BY 1,2
;
```

Are sales at women's clothing stores uniformly higher than those at men's clothing stores? In the yearly trend shown in Figure 3-6, the gap between men's and women's sales does not appear constant but rather was increasing during the early to mid-2000s. Women's clothing sales in particular dipped during the global financial crisis of 2008–2009, and sales in both categories dropped a lot during the pandemic in 2020.

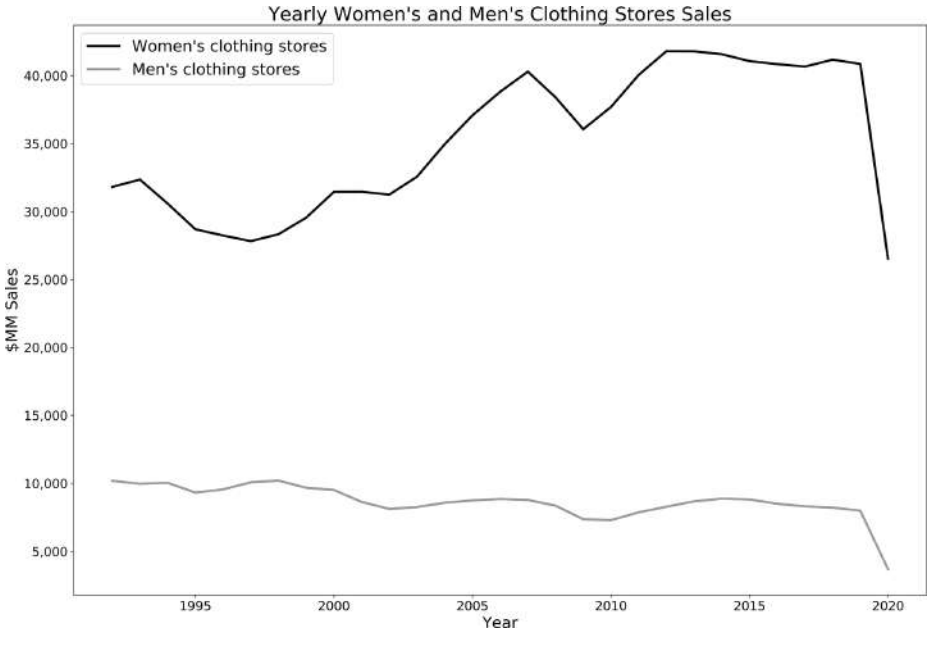


Figure 3-6. Yearly trend of sales at women's and men's clothing stores

We don't need to rely on visual estimation, however. For more precision on this gap, we can calculate the gap between the two categories, the ratio, and the percent difference between them. To do this, the first step is to arrange the data so that there is a single row for each month, with a column for each category. Pivoting the data with aggregate functions combined with CASE statements accomplishes this:

```
SELECT date_part('year',sales_month) as sales_year
,sum(case when kind_of_business = 'Women''s clothing stores'
          then sales
          end) as womens_sales
,sum(case when kind_of_business = 'Men''s clothing stores'
          then sales
          end) as mens_sales
FROM retail_sales
WHERE kind_of_business in ('Men''s clothing stores'
 ,'Women''s clothing stores')
GROUP BY 1
;

sales_year  womens_sales  mens_sales
----------  ------------  ----------
1992.0      31815         10179
1993.0      32350         9962
1994.0      30585         10032
...         ...           ...
```

With this building block calculation, we can find the difference, ratio, and percent difference between time series in the data set. The difference can be calculated by subtracting one value from the other using the mathematical "–" operator. Depending on the goals of the analysis, either finding the difference from men's sales or finding the difference from women's sales might be appropriate. Both are shown here and are equivalent except for the sign:

```
SELECT sales_year
,womens_sales - mens_sales as womens_minus_mens
,mens_sales - womens_sales as mens_minus_womens
FROM
(
    SELECT date_part('year',sales_month) as sales_year
    ,sum(case when kind_of_business = 'Women''s clothing stores'
              then sales
              end) as womens_sales
    ,sum(case when kind_of_business = 'Men''s clothing stores'
              then sales
              end) as mens_sales
    FROM retail_sales
    WHERE kind_of_business in ('Men''s clothing stores'
     ,'Women''s clothing stores')
    and sales_month <= '2019-12-01'
    GROUP BY 1
) a
;

sales_year  womens_minus_mens  mens_minus_womens
----------  -----------------  -----------------
1992.0      21636              -21636
1993.0      22388              -22388
1994.0      20553              -20553
...         ...                ...
```

The subquery is not required from a query execution standpoint, since aggregations can be added to or subtracted from each other. A subquery is often more legible but does add more lines to the code. Depending on how long or complex the rest of your SQL query is, you might prefer to place the intermediate calculation in a subquery, or just calculate it in the main query. Here is an example without the subquery, subtracting men's sales from women's sales, with an added *WHERE* clause filter to remove 2020, since a few months have null values:[1]

```
SELECT date_part('year',sales_month) as sales_year
,sum(case when kind_of_business = 'Women''s clothing stores'
          then sales end)
```

1 October and November 2020 data points were suppressed by the publisher of the data, due to concerns about the data quality. Collecting the data likely became more difficult due to store closures during the 2020 pandemic.

```
 -
 sum(case when kind_of_business = 'Men''s clothing stores'
          then sales end)
 as womens_minus_mens
FROM retail_sales
WHERE kind_of_business in ('Men''s clothing stores'
 ,'Women''s clothing stores')
and sales_month <= '2019-12-01'
GROUP BY 1
;

sales_year  womens_minus_mens
----------  -----------------
1992.0      21636
1993.0      22388
1994.0      20553
...         ...
```

Figure 3-7 shows that the gap decreased between 1992 and about 1997, began a long increase through about 2011 (with a brief dip in 2007), and then was more or less flat through 2019.

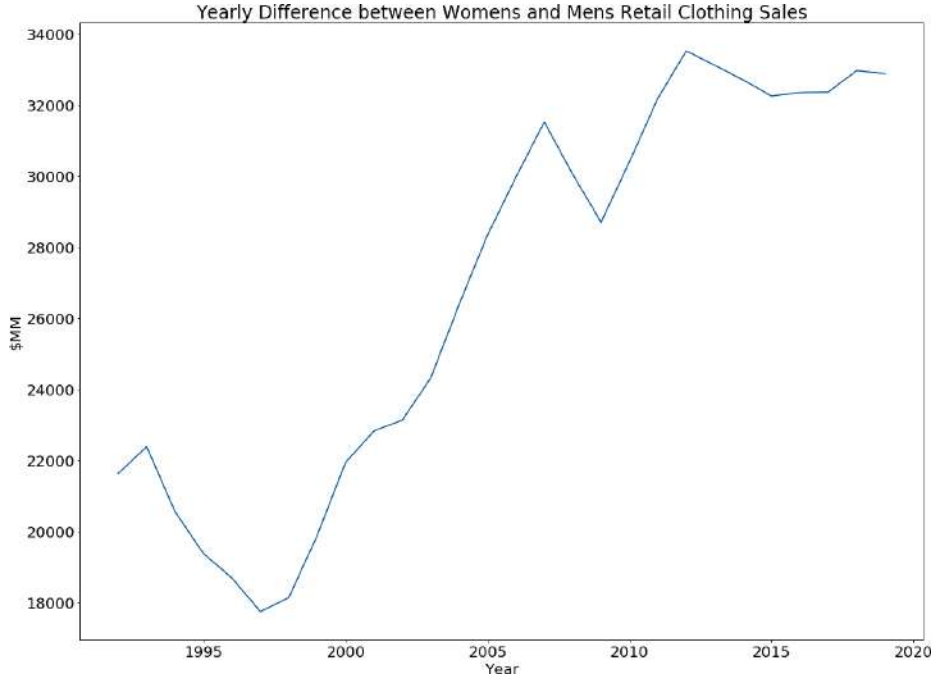


Figure 3-7. Yearly difference between sales at women's and men's clothing stores

Let's continue our investigation and look at the ratio of these categories. We'll use men's sales as the baseline or denominator, but note that we could just as easily use women's store sales instead:

```
SELECT sales_year
,womens_sales / mens_sales as womens_times_of_mens
FROM
(
    SELECT date_part('year',sales_month) as sales_year
    ,sum(case when kind_of_business = 'Women''s clothing stores'
              then sales
              end) as womens_sales
    ,sum(case when kind_of_business = 'Men''s clothing stores'
              then sales
              end) as mens_sales
    FROM retail_sales
    WHERE kind_of_business in ('Men''s clothing stores'
     ,'Women''s clothing stores')
    and sales_month <= '2019-12-01'
    GROUP BY 1
) a
;

sales_year  womens_times_of_mens
----------  --------------------
1992.0      3.1255526083112290
1993.0      3.2473398915880345
1994.0      3.0487440191387560
...         ...
```

SQL returns a lot of decimal digits when performing division. You should generally consider rounding the result before presenting the analysis. Use the level of precision (number of decimal places) that tells the story.

Plotting the result, shown in Figure 3-8, reveals that the trend is similar to the difference trend, but while there was a drop in the difference in 2009, the ratio actually increased.

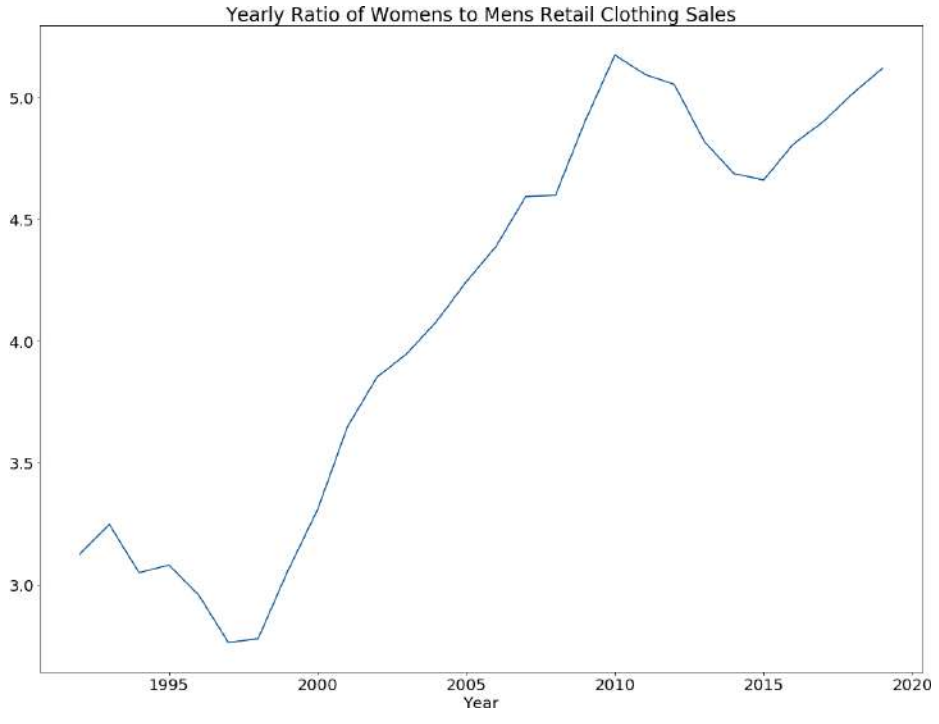


Figure 3-8. Yearly ratio of women's to men's clothing sales

Next, we can calculate the percent difference between sales at women's and men's clothing stores:

```
SELECT sales_year
,(womens_sales / mens_sales - 1) * 100 as womens_pct_of_mens
FROM
(
    SELECT date_part('year',sales_month) as sales_year
    ,sum(case when kind_of_business = 'Women''s clothing stores'
              then sales
              end) as womens_sales
    ,sum(case when kind_of_business = 'Men''s clothing stores'
              then sales
              end) as mens_sales
    FROM retail_sales
    WHERE kind_of_business in ('Men''s clothing stores'
     ,'Women''s clothing stores')
    and sales_month <= '2019-12-01'
    GROUP BY 1
) a
;
```

```
sales_year  womens_pct_of_mens
----------  --------------------
1992.0      212.5552608311229000
1993.0      224.7339891588034500
1994.0      204.8744019138756000
...         ...
```

Although the units for this output are different from those in the previous example, the shape of this graph is the same as that of the ratio graph. The choice of which to use depends on your audience and the norms in your domain. All of these statements are accurate: in 2009, sales at women's clothing stores were $28.7 billion higher than sales at men's stores; in 2009, sales at women's clothing stores were 4.9 times the sales at men's stores; in 2009, sales at women's stores were 390% higher than sales at men's stores. Which version to select depends on the story you want to tell with the analysis.

The transformations we've seen in this section allow us to analyze time series by comparing related parts. The next section will continue the theme of comparing time series by showing ways to analyze series that represent parts of a whole.

Percent of Total Calculations

When working with time series data that has multiple parts or attributes that constitute a whole, it's often useful to analyze each part's contribution to the whole and whether that has changed over time. Unless the data already contains a time series of the total values, we'll need to calculate the overall total in order to calculate the percent of total for each row. This can be accomplished with a self-*JOIN*, or a window function, which as we saw in Chapter 2 is a special kind of SQL function that can reference any row within a specified partition of the table.

First I'll show the self-*JOIN* method. A self-*JOIN* is any time a table is joined to itself. As long as each instance of the table in the query is given a different alias, the database will treat them all as distinct tables. For example, to find the percent of combined men's and women's clothing sales that each series represents, we can *JOIN* `retail_sales`, aliased as `a`, to `retail_sales`, aliased as `b`, on the `sales_month` field. We then *SELECT* the individual series name (`kind_of_business`) and `sales` values from alias `a`. Then, from alias `b` we `sum` the `sales` for both categories and call the result `total_sales`. Note that the *JOIN* between the tables on the `sales_month` field creates a partial Cartesian *JOIN*, which results in two rows from alias `b` for each row in alias `a`. Grouping by `a.sales_month`, `a.kind_of_business`, and `a.sales` and aggregating `b.sales` returns exactly the results needed, however. In the outer query, the percent of total for each row is calculated by dividing `sales` by `total_sales`:

```
SELECT sales_month
,kind_of_business
,sales * 100 / total_sales as pct_total_sales
```

```
FROM
(
    SELECT a.sales_month, a.kind_of_business, a.sales
    ,sum(b.sales) as total_sales
    FROM retail_sales a
    JOIN retail_sales b on a.sales_month = b.sales_month
    and b.kind_of_business in ('Men''s clothing stores'
     ,'Women''s clothing stores')
    WHERE a.kind_of_business in ('Men''s clothing stores'
     ,'Women''s clothing stores')
    GROUP BY 1,2,3
) aa
;

sales_month  kind_of_business         pct_total_sales
-----------  ----------------------  ------------------
1992-01-01   Men's clothing stores    27.2338772338772339
1992-01-01   Women's clothing stores  72.7661227661227661
1992-02-01   Men's clothing stores    24.8395620989052473
...          ...                     ...
```

The subquery isn't required here, as the same result could be obtained without it, but it makes the code a little easier to follow. A second way to calculate the percent of total sales for each category is to use the `sum` window function and *PARTITION BY* the `sales_month`. Recall that the *PARTITION BY* clause indicates the section of the table within which the function should calculate. The *ORDER BY* clause is not required in this `sum` window function, because the order of calculation doesn't matter. Additionally, the query does not need a *GROUP BY* clause, because window functions look across multiple rows, but they do not reduce the number of rows in the result set:

```
SELECT sales_month, kind_of_business, sales
,sum(sales) over (partition by sales_month) as total_sales
,sales * 100 / sum(sales) over (partition by sales_month) as pct_total
FROM retail_sales
WHERE kind_of_business in ('Men''s clothing stores'
 ,'Women''s clothing stores')
;

sales_month  kind_of_business         sales  total_sales  pct_total
-----------  ----------------------  -----  -----------  ---------
1992-01-01   Men's clothing stores    701    2574         27.233877
1992-01-01   Women's clothing stores  1873   2574         72.766122
1992-02-01   Women's clothing stores  1991   2649         75.160437
...          ...                     ...    ...          ...
```

Graphing this data, as in Figure 3-9, reveals some interesting trends. First, starting in the late 1990s, women's clothing store sales became an increasing percentage of the total. Second, early in the series a seasonal pattern is evident, where men's sales spike as a percent of total sales in December and January. In the first decade of the 21st

century, two seasonal peaks appear, in the summer and the winter, but by the late 2010s, the seasonal patterns are dampened almost to the point of randomness. We'll take a look at analyzing seasonality in greater depth later in this chapter.

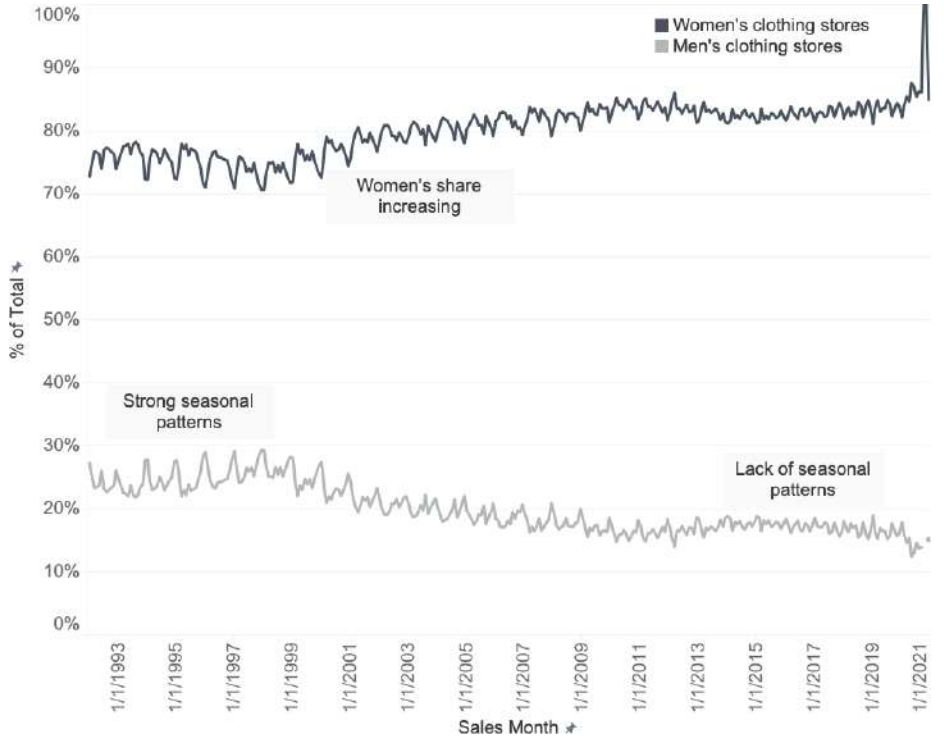


Figure 3-9. Men's and women's clothing store sales as percent of monthly total

Another percent of total we might want to find is the percent of sales within a longer time period, such as the percent of yearly sales each month represents. Again, either a self-*JOIN* or a window function will do the job. In this example, we'll use a self-*JOIN* in the subquery:

```
SELECT sales_month
,kind_of_business
,sales * 100 / yearly_sales as pct_yearly
FROM
(
    SELECT a.sales_month, a.kind_of_business, a.sales
    ,sum(b.sales) as yearly_sales
    FROM retail_sales a
    JOIN retail_sales b on
     date_part('year',a.sales_month) = date_part('year',b.sales_month)
     and a.kind_of_business = b.kind_of_business
     and b.kind_of_business in ('Men''s clothing stores'
```

```
      ,'Women''s clothing stores')
    WHERE a.kind_of_business in ('Men''s clothing stores'
     ,'Women''s clothing stores')
    GROUP BY 1,2,3
) aa
;

sales_month  kind_of_business       pct_yearly
-----------  --------------------  ------------------
1992-01-01   Men's clothing stores  6.8867275763827488
1992-02-01   Men's clothing stores  6.4642892229099126
1992-03-01   Men's clothing stores  7.1814520090382159
...          ...                   ...
```

Alternatively, the window function method can be used:

```
SELECT sales_month, kind_of_business, sales
,sum(sales) over (partition by date_part('year',sales_month)
                               ,kind_of_business
                               ) as yearly_sales
,sales * 100 /
 sum(sales) over (partition by date_part('year',sales_month)
                               ,kind_of_business
                               ) as pct_yearly
FROM retail_sales
WHERE kind_of_business in ('Men''s clothing stores'
 ,'Women''s clothing stores')
;

sales_month  kind_of_business       pct_yearly
-----------  --------------------  ------------------
1992-01-01   Men's clothing stores  6.8867275763827488
1992-02-01   Men's clothing stores  6.4642892229099126
1992-03-01   Men's clothing stores  7.1814520090382159
...          ...                   ...
```

The results, zoomed in to 2019, are shown in Figure 3-10. The two time series track fairly closely, but men's stores had a greater percentage of their sales in January than did women's stores. Men's stores had a summer dip in July, while the corresponding dip in women's store sales wasn't until September.

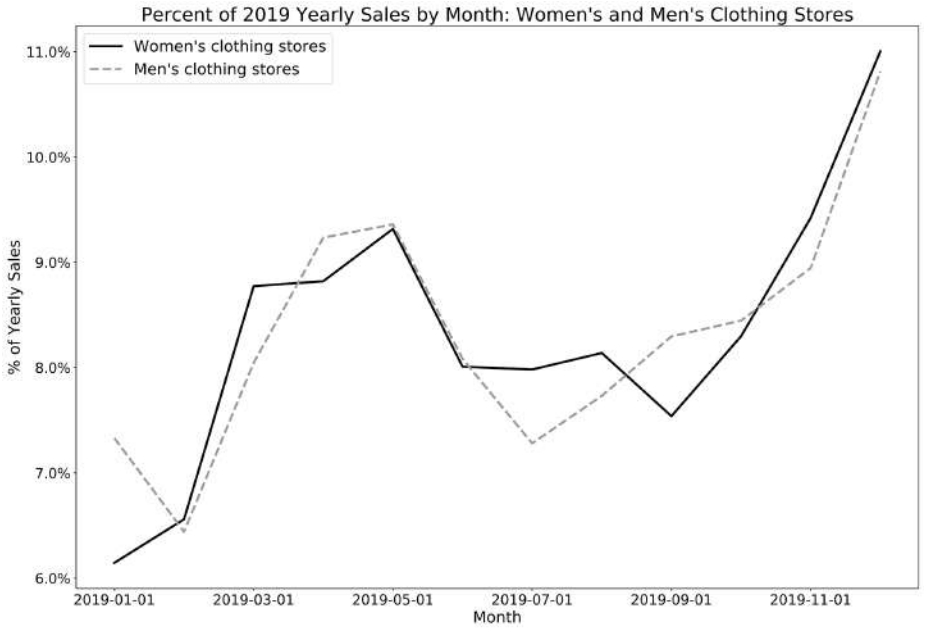


Figure 3-10. Percent of yearly sales for 2019 for women's and men's clothing sales

Now that I've shown how to use SQL for percent of total calculations and the types of analysis that can be accomplished, I'll turn to indexing and calculating percent change over time.

Indexing to See Percent Change over Time

The values in time series usually fluctuate over time. Sales increase with growing popularity and availability of a product, while web page response time decreases with engineers' efforts to optimize code. Indexing data is a way to understand the changes in a time series relative to a base period (starting point). Indices are widely used in economics as well as business settings. One of the most famous indices is the Consumer Price Index (CPI), which tracks the change in the prices of items that a typical consumer purchases and is used to track inflation, to decide salary increases, and for many other applications. The CPI is a complex statistical measure using various weights and data inputs, but the basic premise is straightforward. Pick a base period and compute the percent change in value from that base period for each subsequent period.

Indexing time series data with SQL can be done with a combination of aggregations and window functions, or self-*JOIN*s. As an example, we index women's clothing store sales to the first year in the series, 1992. The first step is to aggregate the `sales` by `sales_year` in a subquery, as we've done previously. In the outer query, the `first_value` window function finds the value associated with the first row in the *PARTITION BY* clause, according to the sort in the *ORDER BY* clause. In this example, we can omit the *PARTITION BY* clause, because we want to return the sales `value` for the first row in the entire data set returned by the subquery:

```
SELECT sales_year, sales
,first_value(sales) over (order by sales_year) as index_sales
FROM
(
    SELECT date_part('year',sales_month) as sales_year
    ,sum(sales) as sales
    FROM retail_sales
    WHERE kind_of_business = 'Women''s clothing stores'
    GROUP BY 1
) a
;

sales_year  sales  index_sales
----------  -----  -----------
1992.0      31815  31815
1993.0      32350  31815
1994.0      30585  31815
...         ...    ...
```

With this sample of data, we can visually verify that the index value is correctly set at the value for 1992. Next, find the percent change from this base year for each row:

```
SELECT sales_year, sales
,(sales / first_value(sales) over (order by sales_year) - 1) * 100
 as pct_from_index
FROM
(
    SELECT date_part('year',sales_month) as sales_year
    ,sum(sales) as sales
    FROM retail_sales
    WHERE kind_of_business = 'Women''s clothing stores'
    GROUP BY 1
) a
;

sales_year  sales  pct_from_index
----------  -----  --------------
1992.0      31815  0
1993.0      32350  1.681596731101
1994.0      30585  -3.86610089580
...         ...    ...
```

The percent change can be either positive or negative, and we'll see that does in fact occur in this time series. The `last_value` window function could be substituted for `first_value` in this query. Indexing from the last value in a series is much less common, however, since analysis questions more often relate to change from a starting point rather than looking back from an arbitrary ending point; still, the option is there. Additionally, the sort order can be used to achieve indexing from the first or last value by switching between *ASC* and *DESC*:

```
first_value(sales) over (order by sales_year desc)
```

Window functions provide a lot of flexibility. Indexing can be accomplished without them through a series of self-*JOIN*s, though more lines of code are required:

```
SELECT sales_year, sales
,(sales / index_sales - 1) * 100 as pct_from_index
FROM
(
    SELECT date_part('year',aa.sales_month) as sales_year
    ,bb.index_sales
    ,sum(aa.sales) as sales
    FROM retail_sales aa
    JOIN
    (
        SELECT first_year, sum(a.sales) as index_sales
        FROM retail_sales a
        JOIN
        (
            SELECT min(date_part('year',sales_month)) as first_year
            FROM retail_sales
            WHERE kind_of_business = 'Women''s clothing stores'
        ) b on date_part('year',a.sales_month) = b.first_year
        WHERE a.kind_of_business = 'Women''s clothing stores'
        GROUP BY 1
    ) bb on 1 = 1
    WHERE aa.kind_of_business = 'Women''s clothing stores'
    GROUP BY 1,2
) aaa
;

sales_year  sales  pct_from_index
----------  -----  --------------
1992.0      31815  0
1993.0      32350  1.681596731101
1994.0      30585  -3.86610089580
...         ...    ...
```

Notice the unusual *JOIN* clause `on 1 = 1` between alias `aa` and subquery `bb`. Since we want the `index_sales` value to populate for every row in the result set, we can't *JOIN* on the year or any other value, which would restrict the results. However, the database will return an error if no *JOIN* clause is specified. We can fool the database by using any expression that evaluates to TRUE in order to create the desired Cartesian *JOIN*. Any other TRUE statement, such as `on 2 = 2` or `on 'apples' = 'apples'`, could be used instead.

Beware of zeros in the denominator of division operations such as `sales / index_sales` in the last example. Databases return an error when they encounter division by zero, which can be frustrating. Even when you think a zero in the denominator field is unlikely, it's good practice to prevent this by telling the database to return an alternate default value when it encounters a zero. This can be done with a CASE statement. The examples in this section do not have zeros in the denominator, so I will omit this extra code for the sake of legibility.

To wrap up this section, let's look at a graph of the indexed time series for men's and women's clothing stores, shown in Figure 3-11. The SQL code looks like:

```
SELECT sales_year, kind_of_business, sales
,(sales / first_value(sales) over (partition by kind_of_business
                                   order by sales_year)
 - 1) * 100 as pct_from_index
FROM
(
    SELECT date_part('year',sales_month) as sales_year
    ,kind_of_business
    ,sum(sales) as sales
    FROM retail_sales
    WHERE kind_of_business in ('Men''s clothing stores'
     ,'Women''s clothing stores')
    and sales_month <= '2019-12-31'
    GROUP BY 1,2
) a
;
```

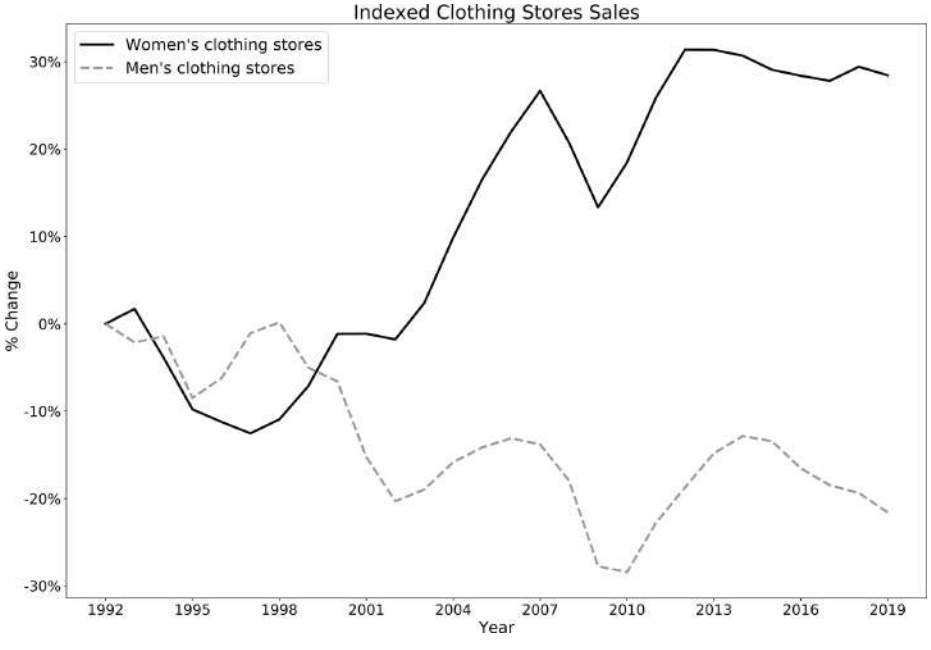


Figure 3-11. Men's and women's clothing store sales, indexed to 1992 sales

It's apparent from this graph that 1992 was something of a high-water mark for sales at men's clothing stores. After 1992 sales dropped, then returned briefly to the same level in 1998, and have been declining ever since. This is striking since the data set is not adjusted for inflation, the tendency for prices to rise over time. Sales at women's clothing stores decreased from 1992 levels initially, but they returned to the 1992 level by 2003. They have increased since, with the exception of the drop during the financial crisis that decreased sales in 2009 and 2010. One explanation for these trends is that men simply decreased spending on clothes over time, perhaps becoming less fashion conscious relative to women. Perhaps men's clothing simply became less expensive as global supply chains decreased costs. Yet another explanation might be that men shifted their clothing purchases from retailers categorized as "men's clothing stores" to other types of retailers, such as sporting goods stores or online retailers.

Indexing time series data is a powerful analysis technique, allowing us to find a range of insights in the data. SQL is well suited to this task, and I've shown how to construct indexed time series with and without window functions. Next, I'll show you how to analyze data by using rolling time windows to find patterns in noisy time series.

Rolling Time Windows

Time series data is often noisy, a challenge for one of our primary goals of finding patterns. We've seen how aggregating data, such as from monthly to yearly, can smooth out the results and make them easier to interpret. Another technique for smoothing data is *rolling time windows*, also known as moving calculations, that take into account multiple periods. Moving averages are probably the most common, but with the power of SQL, any aggregate function is available for analysis. Rolling time windows are used in a wide variety of analysis areas, including stock markets, macroeconomic trends, and audience measurement. Some calculations are so commonly used that they have their own acronyms: last twelve months (LTM), trailing twelve months (TTM), and year-to-date (YTD).

Figure 3-12 shows an example of a rolling time window and a cumulative calculation, relative to the month of October in the time series.

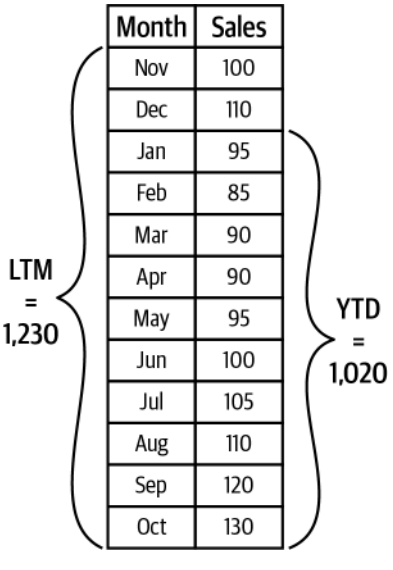


Figure 3-12. Example of LTM and YTD rolling sum of sales

There are several important pieces of any rolling time series calculation. First is the size of the window, which is the number of periods to include in the calculation. Larger windows with more time periods have a greater smoothing effect, but at the risk of losing sensitivity to important short-term changes in the data. Shorter windows with fewer time periods do less smoothing and thus are more sensitive to short-term changes, but at the risk of too little noise reduction.

The second piece of time series calculations is the aggregate function used. As noted previously, moving averages are probably the most common. Moving sums, counts,

minimums, and maximums can also be calculated with SQL. Moving counts are useful in user population metrics (see the following sidebar). Moving minimums and maximums can help in understanding the extremes of the data, useful for planning analyses.

The third piece of time series calculations is choosing the partitioning, or grouping, of the data that is included in the window. The analysis might call for resetting the window every year. Or the analysis might need a different moving series for each component or user group. Chapter 4 will go into more detail on cohort analysis of user groups, where we will consider how retention and cumulative values such as spend differ between populations over time. Partitioning will be controlled through grouping as well as the *PARTITION BY* statement of window functions.

With these three pieces in mind, we'll move into the SQL code and calculations for moving time periods, continuing with the US retail sales data set for examples.

Measuring "Active Users": DAU, WAU, and MAU

Many consumer and some B2B SaaS applications use active user calculations such as daily active users (DAU), weekly active users (WAU), and monthly active users (MAU) to estimate their audience size. Since each of these are rolling windows, they can be calculated on a daily basis. I've often been asked what is the right or best metric to use, and my answer is always "it depends."

DAU helps companies with capacity planning, such as estimating how much load to expect on servers. Depending on the service, however, even more detailed data might be needed, such as peak hourly or even minute-by-minute concurrent user information.

MAU is commonly used to estimate relative sizes of applications or services. It is useful for measuring fairly stable or growing user populations that have regular usage patterns that aren't necessarily daily, such as higher use on the weekend for leisure products, or higher weekday use for work- or school-related products. MAU is not as well suited to detecting changes in underlying churn from users who stop using an application. Since it takes a user 30 days, the most common window, to pass through MAU, a user can have been absent from the product for 29 days before they trigger a drop in MAU.

WAU, calculated over 7 days, can be a happy medium between DAU and MAU. WAU is more sensitive to short-term fluctuations, alerting teams to changes in churn more quickly than MAU while smoothing over day of week fluctuations that are tracked by DAU. A drawback to WAU is that it is still sensitive to short-term fluctuations driven by events such as holidays.

Calculating Rolling Time Windows

Now that we know what rolling time windows are, how they're useful, and their key components, let's get into calculating them using the US retail sales data set. We'll start with the simpler case, when the data set contains a record for each period that should be in the window, and then in the next section we'll look at what to do when this is not the case.

There are two main methods for calculating a rolling time window: a self-*JOIN*, which can be used in any database, and a window function, which as we've seen isn't available in some databases. In both cases we need the same result: a date and a number of data points that corresponds to the size of the window to which we will apply an average or another aggregate function.

For this example, we'll use a window of 12 months to get rolling annual sales, since the data is at a monthly level of granularity. We'll then apply an average to get a 12-month moving average of retail sales. First, let's develop the intuition for what will go into the calculation. In this query, alias `a` of the table is our "anchor" table, the one from which we gather the dates. To start, we'll look at a single month, December 2019. From alias `b`, the query gathers the 12 individual months of sales that will go into the moving average. This is accomplished with the *JOIN* clause `b.sales_month between a.sales_month - interval '11 months' and a.sales_month`, which creates an intentional Cartesian *JOIN*:

```
SELECT a.sales_month
,a.sales
,b.sales_month as rolling_sales_month
,b.sales as rolling_sales
FROM retail_sales a
JOIN retail_sales b on a.kind_of_business = b.kind_of_business
 and b.sales_month between a.sales_month - interval '11 months'
 and a.sales_month
 and b.kind_of_business = 'Women''s clothing stores'
WHERE a.kind_of_business = 'Women''s clothing stores'
and a.sales_month = '2019-12-01'
;

sales_month  sales  rolling_sales_month  rolling_sales
-----------  -----  -------------------  -------------
2019-12-01   4496   2019-01-01           2511
2019-12-01   4496   2019-02-01           2680
2019-12-01   4496   2019-03-01           3585
2019-12-01   4496   2019-04-01           3604
2019-12-01   4496   2019-05-01           3807
2019-12-01   4496   2019-06-01           3272
2019-12-01   4496   2019-07-01           3261
2019-12-01   4496   2019-08-01           3325
2019-12-01   4496   2019-09-01           3080
2019-12-01   4496   2019-10-01           3390
```

```
2019-12-01   4496   2019-11-01          3850
2019-12-01   4496   2019-12-01          4496
```

Notice that the `sales_month` and `sales` figures from alias `a` are repeated for each row of the 12 months in the window.

Remember that the dates in a *BETWEEN* clause are inclusive (both will be returned in the result set). It's a common mistake to use 12 instead of 11 in the preceding query. When in doubt, check the intermediate query results as I've done here to make sure the intended number of periods ends up in the window calculation.

The next step is to apply the aggregation—in this case, `avg`, since we want a rolling average. The `count` of records returned from alias `b` is included to confirm that each row averages 12 data points, a useful data quality check. Alias `a` also has a filter on `sales_month`. Since this data set starts in 1992, months in that year, except for December, have fewer than 12 historical records:

```
SELECT a.sales_month
,a.sales
,avg(b.sales) as moving_avg
,count(b.sales) as records_count
FROM retail_sales a
JOIN retail_sales b on a.kind_of_business = b.kind_of_business
 and b.sales_month between a.sales_month - interval '11 months'
 and a.sales_month
 and b.kind_of_business = 'Women''s clothing stores'
WHERE a.kind_of_business = 'Women''s clothing stores'
and a.sales_month >= '1993-01-01'
GROUP BY 1,2
;


sales_month  sales  moving_avg  records_count
-----------  -----  ----------  -------------
1993-01-01   2123   2672.08     12
1993-02-01   2005   2673.25     12
1993-03-01   2442   2676.50     12
...          ...    ...         ...
```

The results are graphed in Figure 3-13. While the monthly trend is noisy, the smoothed moving average trend makes detecting changes such as the increase from 2003 to 2007 and the subsequent dip through 2011 easier to spot. Notice that the extreme drop in early 2020 pulls the moving average down even after sales start to rebound later in the year.

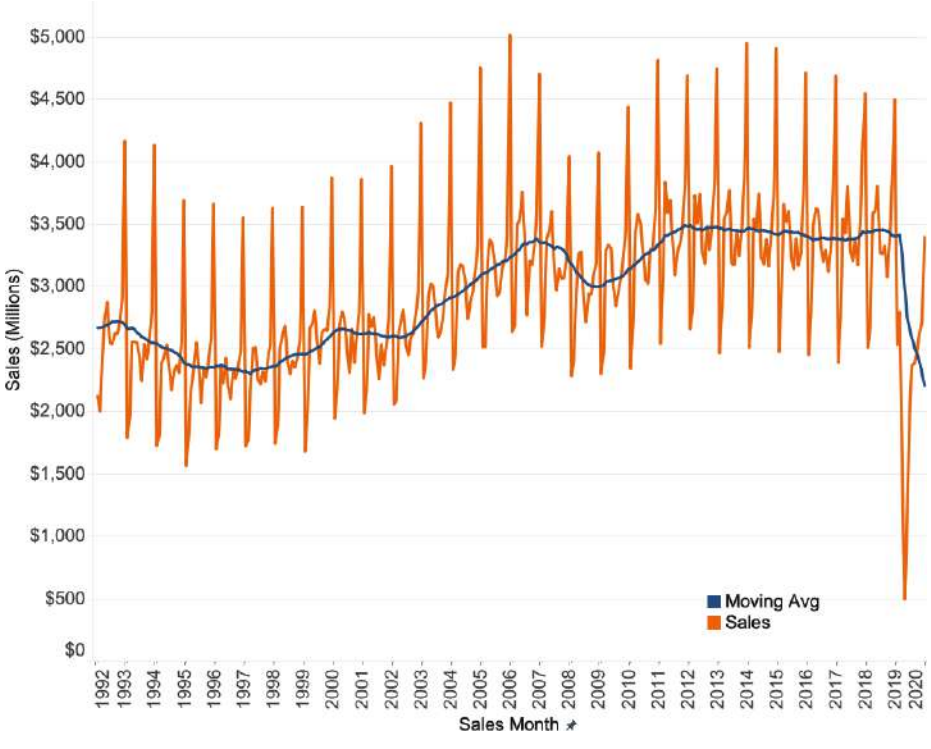


Figure 3-13. Monthly sales and 12-month moving average sales for women's clothing stores

Adding the filter `kind_of_business = 'Women''s clothing stores'` to each alias isn't strictly necessary. Since the query uses an *INNER JOIN*, filtering on one table will automatically filter on the other. However, filtering on both tables often makes queries run faster, particularly when the tables are large.

Window functions are another way to calculate rolling time windows. To make a rolling window, we need to use another optional part of a window calculation: the *frame clause*. The frame clause allows you to specify which records to include in the window. By default, all records in the partition are included, and for many cases this works just fine. However, controlling the included records at a more fine-grained level is useful for cases like moving window calculations. The syntax is simple and yet can be confusing when encountering it for the first time. The frame clause can be specified as:

```
{ RANGE | ROWS | GROUPS } BETWEEN frame_start AND frame_end
```

Within the curly braces are three options for the frame type: range, rows, and groups. These are the ways you can specify which records to include in the result, relative to the current row. Records are always chosen from the current partition and follow the *ORDER BY* specified. The default sorting is ascending (*ASC*), but it can be changed to descending (*DESC*). *Rows* is the most straightforward and will allow you to specify the exact number of rows that should be returned. *Range* includes records that are within some boundary of values relative to the current row. *Groups* can be used when there are multiple records with the same *ORDER BY* value, such as when a data set includes multiple lines per sales month, one for each customer.

The *frame_start* and *frame_end* can be any of the following:

```
UNBOUNDED PRECEDING
offset PRECEDING
CURRENT ROW
offset FOLLOWING
UNBOUNDED FOLLOWING
```

Preceding means to include rows before the current row, according to the *ORDER BY* sorting. *Current row* is just that, and *following* means to include rows that occur after the current row according to the *ORDER BY* sorting. The *UNBOUNDED* keyword means to include all records in the partition before or after the current row. The *offset* is the number of records, often just an integer constant, though a field or an expression that returns an integer could also be used. Frame clauses also have an optional *frame_exclusion* option, which is beyond the scope of the discussion here. Figure 3-14 shows an example of the rows that each of the window frame options will pick up.

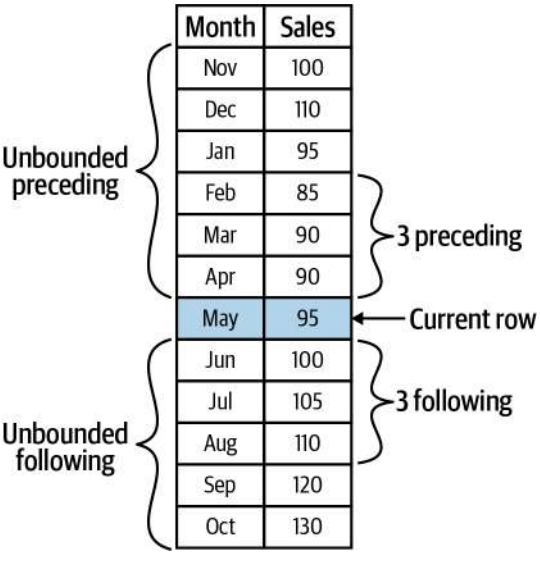


Figure 3-14. Window frame clauses and the rows they include

From partition to ordering to window frames, window functions have a variety of options that control the calculations, making them incredibly powerful and well suited to tackling complex calculations with relatively simple syntax. Returning to our retail sales example, the moving average that we calculated using a self-*JOIN* can be accomplished with window functions in fewer lines of code:

```
SELECT sales_month
,avg(sales) over (order by sales_month
                  rows between 11 preceding and current row
                  ) as moving_avg
,count(sales) over (order by sales_month
                  rows between 11 preceding and current row
                  ) as records_count
FROM retail_sales
WHERE kind_of_business = 'Women''s clothing stores'
;

sales_month  moving_avg  records_count
-----------  ----------  -------------
1992-01-01   1873.00     1
1992-02-01   1932.00     2
1992-03-01   2089.00     3
...          ...         ...
1993-01-01   2672.08     12
1993-02-01   2673.25     12
1993-03-01   2676.50     12
...          ...         ...
```

In this query, the window orders the sales by month (ascending) to ensure that the window records are in chronological order. The frame clause is `rows between 11 preceding and current row`, since I know that I have one record for each month and I want the 11 prior months and the month from the current row included in the average and count calculations. The query returns all months, including those that don't have 11 prior months, and we might want to filter these out by placing this query in a subquery and filtering by month or number of records in the outer query.

While calculating moving averages from prior time periods is common in many business contexts, SQL window functions are flexible enough to include future time periods as well. They can also be used in any scenario in which the data has some ordering, not just in time series analysis.

Calculating rolling averages or other moving aggregations can be accomplished with self-*JOIN*s or window functions when records exist in the data set for each time period in the window. There may be performance differences between the two methods, depending on the type of database and the size of the data set. Unfortunately, it's difficult to predict which one will be performant or to give general advice on which to

use. It's worth trying both methods and paying attention to how long it takes to return your query results; then make whichever one seems to run faster your default choice. Now that we've seen how to calculate rolling time windows, I'll show how to calculate rolling windows with sparse data sets.

Rolling Time Windows with Sparse Data

Data sets in the real world may not contain a record for every time period that falls within the window. The measurement of interest might be seasonal or intermittent by nature. For example, customers might return to purchase from a website at irregular intervals, or a particular product might go in and out of stock. This results in sparse data.

In the last section, I showed how to calculate a rolling window with a self-*JOIN* and a date interval in the *JOIN* clause. You might be thinking that this will pick up any records within the 12-month time window, whether all were in the data set or not, and you'd be correct. The problem with this approach comes when there is no record for the month (or day or year) itself. For example, imagine I want to calculate the rolling 12-month sales for each model of shoe my store stocks as of December 2019. Some of the shoes went out of stock prior to December, however, and so don't have sales records in that month. Using a self-*JOIN* or window function will return a data set of rolling sales for all the shoes that sold in December, but the data will be missing the shoes that went out of stock. Fortunately, we have a way to solve this problem: by using a date dimension.

The *date dimension*, a static table that contains a row for each calendar date, was introduced in Chapter 2. With such a table we can ensure that a query returns a result for every date of interest, whether or not there was a data point for that date in the underlying data set. Since the `retail_sales` data does include rows for all months, I've simulated a sparse data set by adding a subquery to filter the table to only `sales_month`s from January and July (1 and 7). Let's look at the results when *JOIN*ed to the `date_dim`, but before aggregation, to develop intuition about the data before applying calculations:

```
SELECT a.date, b.sales_month, b.sales
FROM date_dim a
JOIN
(
    SELECT sales_month, sales
    FROM retail_sales
    WHERE kind_of_business = 'Women''s clothing stores'
     and date_part('month',sales_month) in (1,7)
) b on b.sales_month between a.date - interval '11 months' and a.date
WHERE a.date = a.first_day_of_month
 and a.date between '1993-01-01' and '2020-12-01'
;
```

```
date        sales_month  sales
----------  -----------  -----
1993-01-01  1992-07-01   2373
1993-01-01  1993-01-01   2123
1993-02-01  1992-07-01   2373
1993-02-01  1993-01-01   2123
1993-03-01  1992-07-01   2373
...         ...          ...
```

Notice that the query returns results for February and March dates in addition to January, even though there are no sales for these months in the subquery results. This is possible because the date dimension contains records for all months. The filter `a.date = a.first_day_of_month` restricts the result set to one value per month, instead of the 28 to 31 rows per month that would result from joining to every date. The construction of this query is otherwise very similar to the self-*JOIN* query in the last section, with the *JOIN* clause `on b.sales_month between a.date - interval '11 months' and a.date` of the same form as the *JOIN* clause in the self-*JOIN*. Now that we have developed an understanding of what the query will return, we can go ahead and apply the `avg` aggregation to get the moving average:

```
SELECT a.date
,avg(b.sales) as moving_avg
,count(b.sales) as records
FROM date_dim a
JOIN
(
    SELECT sales_month, sales
    FROM retail_sales
    WHERE kind_of_business = 'Women''s clothing stores'
     and date_part('month',sales_month) in (1,7)
) b on b.sales_month between a.date - interval '11 months' and a.date
WHERE a.date = a.first_day_of_month
 and a.date between '1993-01-01' and '2020-12-01'
GROUP BY 1
;

date        moving_avg  records
----------  ----------  -------
1993-01-01  2248.00     2
1993-02-01  2248.00     2
1993-03-01  2248.00     2
...         ...         ...
```

As we saw above, the result set includes a row for every month; however, the moving average stays constant until a new data point (in this case, a January or July) is added. Each moving average consists of two underlying data points. In a real use case, the number of underlying data points is likely to vary. To return the current month's value when using a data dimension, an aggregation with a CASE statement can be used—for example:

```
,max(case when a.date = b.sales_month then b.sales end)
 as sales_in_month
```

The conditions inside the CASE statement can be changed to return any of the underlying records that the analysis requires through use of equality, inequality, or offsets with date math. If a date dimension is not available in your database, then another technique can be used to simulate one. In a subquery, *SELECT* the *DISTINCT* dates needed and *JOIN* them to your table in the same way as in the preceding examples:

```
SELECT a.sales_month, avg(b.sales) as moving_avg
FROM
(
    SELECT distinct sales_month
    FROM retail_sales
    WHERE sales_month between '1993-01-01' and '2020-12-01'
) a
JOIN retail_sales b on b.sales_month between
 a.sales_month - interval '11 months' and a.sales_month
 and b.kind_of_business = 'Women''s clothing stores'
GROUP BY 1
;

sales_month  moving_avg
-----------  ----------
1993-01-01   2672.08
1993-02-01   2673.25
1993-03-01   2676.50
...          ...
```

In this example, I used the same underlying table because I know it contains all the months. However, in practice any database table that contains the needed dates can be used, whether or not it is related to the table from which you want to calculate the rolling aggregation.

Calculating rolling time windows with sparse or missing data can be done in SQL with controlled application of Cartesian *JOIN*s. Next, we'll look at how to calculate cumulative values that are often used in analysis.

Calculating Cumulative Values

Rolling window calculations, such as moving averages, typically use fixed-size windows, such as 12 months, as we saw in the last section. Another commonly used type of calculation is the *cumulative value*, such as YTD, quarter-to-date (QTD), and month-to-date (MTD). Rather than a fixed-length window, these rely on a common starting point, with the window size growing with each row.

The simplest way to calculate cumulative values is with a window function. In this example, `sum` is used to find total sales YTD as of each month. Other analyses might call for a monthly average YTD or a monthly maximum YTD, which can be accomplished by swapping `sum` for `avg` or `max`. The window resets according to the *PARTITION BY* clause, in this case the year of the sales month. The *ORDER BY* clause typically includes a date field in time series analysis. Omitting the *ORDER BY* can lead to incorrect results due to the way the data is sorted in the underlying table, so it's a good idea to include it even if you think the data is already sorted by date:

```
SELECT sales_month, sales
,sum(sales) over (partition by date_part('year',sales_month)
                  order by sales_month
                  ) as sales_ytd
FROM retail_sales
WHERE kind_of_business = 'Women''s clothing stores'
;

sales_month  sales  sales_ytd
-----------  -----  ---------
1992-01-01   1873   1873
1992-02-01   1991   3864
1992-03-01   2403   6267
...          ...    ...
1992-12-01   4416   31815
1993-01-01   2123   2123
1993-02-01   2005   4128
...          ...    ...
```

The query returns a record for each `sales_month`, the `sales` for that month, and the running total `sales_ytd`. The series starts in 1992 and then resets in January 1993, as it will for every year in the data set. The results for years 2016 through 2020 are graphed in Figure 3-15. The first four years show similar patterns through the year, but of course 2020 looks very different.

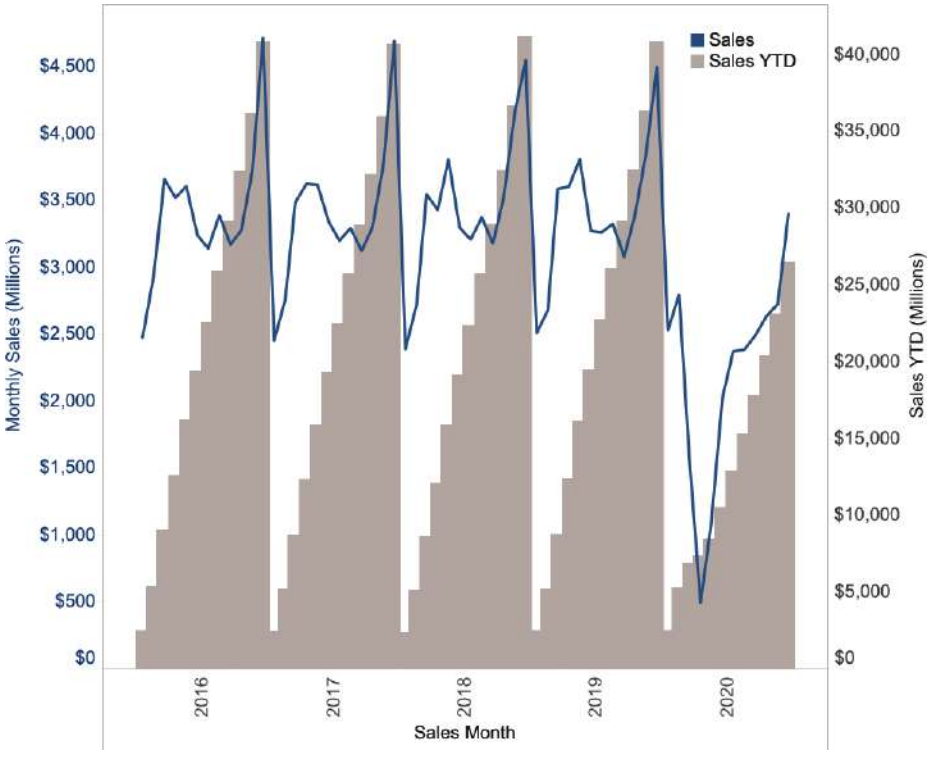


Figure 3-15. Monthly sales and cumulative annual sales for women's clothing stores

The same results can be achieved without window functions, by using a self-*JOIN* that leverages a Cartesian *JOIN*. In this example, the two table aliases are *JOIN*ed on the year of the `sales_month` to ensure that the aggregated values are for the same year, resetting each year. The *JOIN* clause also specifies that the results should include `sales_month`s from alias `b` that are less than or equal to the `sales_month` in alias `a`. In January 1992, only the January 1992 row from alias `b` meets this criterion; in February 1992, both January and February 1992 do; and so on:

```
SELECT a.sales_month, a.sales
,sum(b.sales) as sales_ytd
FROM retail_sales a
JOIN retail_sales b on
 date_part('year',a.sales_month) = date_part('year',b.sales_month)
 and b.sales_month <= a.sales_month
 and b.kind_of_business = 'Women''s clothing stores'
WHERE a.kind_of_business = 'Women''s clothing stores'
GROUP BY 1,2
;
```

```
sales_month  sales  sales_ytd
-----------  -----  ---------
1992-01-01   1873   1873
1992-02-01   1991   3864
1992-03-01   2403   6267
...          ...    ...
1992-12-01   4416   31815
1993-01-01   2123   2123
1993-02-01   2005   4128
...          ...    ...
```

Window functions require fewer characters of code, and it's usually easier to keep track of exactly what they are calculating once you are familiar with the syntax. There's often more than one way to approach a problem in SQL, and rolling time windows are a good example of that. I find it useful to know multiple approaches, because every once in a while I run into a tricky problem that is actually better solved with an approach that seems less efficient in other contexts. Now that we've covered rolling time windows, we'll move on to our final topic in time series analysis with SQL: seasonality.

Analyzing with Seasonality

Seasonality is any pattern that repeats over regular intervals. Unlike other noise in the data, seasonality can be predicted. The word *seasonality* brings to mind the four seasons of the year—spring, summer, fall, winter—and some data sets include these patterns. Shopping patterns change with the seasons, from the clothes and food people buy to the money spent on leisure and travel. The winter holiday shopping season can be make-or-break for many retailers. Seasonality can also exist at other time scales, from years down to minutes. Presidential elections in the United States happen every four years, leading to distinct patterns in media coverage. Day of week cyclicality is common, as work and school dominate Monday to Friday, while chores and leisure activities dominate the weekend. Time of day is another type of seasonality that restaurants experience, with rushes around lunch and dinner time and slower sales in between.

To understand whether seasonality exists in a time series, and at what scale, it's useful to graph it and then visually inspect for patterns. Try aggregating at different levels, from hourly to daily, weekly, and monthly. You should also incorporate knowledge about the data set. Are there patterns that you can guess based on what you know about the entity or process it represents? Consult subject matter experts, if available.

Let's take a look at some seasonal patterns in the retail sales data set, shown in Figure 3-16. Jewelry stores have a highly seasonal pattern, with annual peaks in December related to holiday gift giving. Book stores have two peaks each year: one peak is in August, corresponding with back-to-school time in the United States; the other peak starts in December and lasts through January, including both the holiday

gift period and back-to-school time for the spring semester. A third example is grocery stores, which have much less monthly seasonality than the other two time series (although they likely have seasonality at the day of week and time of day level). This isn't surprising: people need to eat year-round. Grocery store sales increase a bit in December for the holidays, and they decline in February, since that month simply has fewer days.

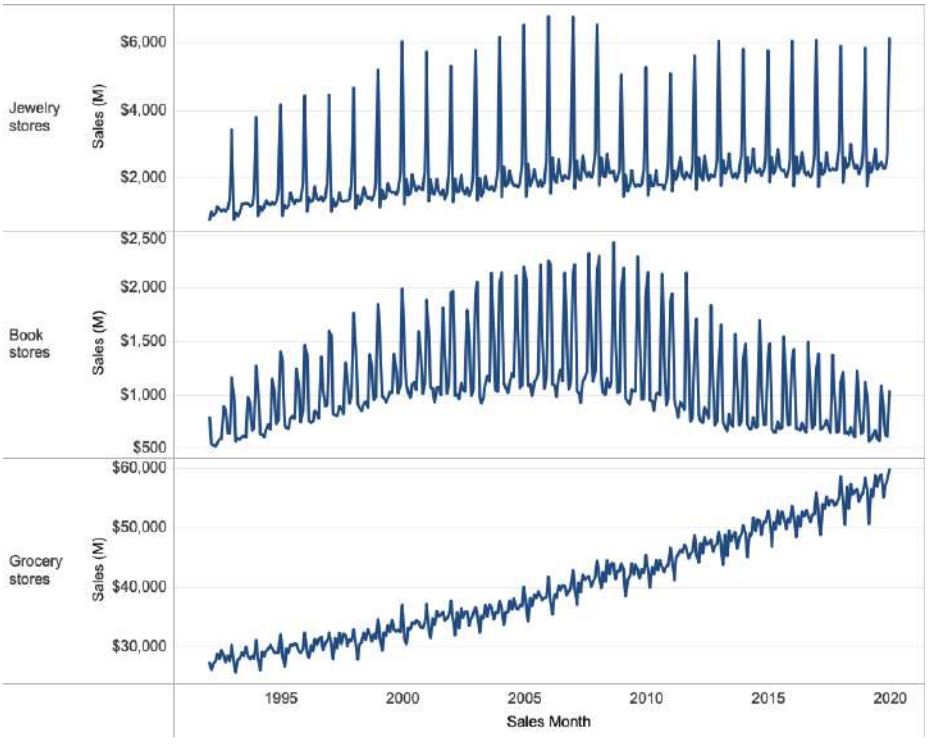


Figure 3-16. Examples of seasonality patterns in book store, grocery store, and jewelry store sales

Seasonality can take many forms, though there are some common approaches to analyzing it regardless. One way to deal with seasonality is to smooth it out, either by aggregating the data to a less granular time period or by using rolling windows, as we saw previously. Another way to work with seasonal data is to benchmark against similar time periods and analyze the difference. I'll show several ways to accomplish this next.

Period-over-Period Comparisons: YoY and MoM

Period-over-period comparisons can take multiple forms. The first one is to compare a time period to the previous value in the series, a practice so common in analysis that there are acronyms for the most often-used comparisons. Depending on the level of aggregation the comparison might be year-over-year (YoY), month-over-month (MoM), day-over-day (DoD), and so on.

For these calculations we'll use the `lag` function, another one of the window functions. The `lag` function returns a previous or lagging value from a series. The `lag` function has the following form:

```
lag(return_value [,offset [,default]])
```

The *`return_value`* is any field from the data set and thus can be any data type. The optional *OFFSET* indicates how many rows back in the partition to take the *`return_value`* from. The default is 1, but any integer value can be used. You can also optionally specify a *`default`* value to use if there is no lagging record to retrieve a value from. Like other window functions, `lag` is also calculated over a partition, with sorting determined by the *ORDER BY* clause. If no *PARTITION BY* clause is specified, `lag` looks back over the whole data set, and likewise if no *ORDER BY* clause is specified, the database order is used. It's usually a good idea to at least include an *ORDER BY* clause in a `lag` window function to control the output.

The `lead` window function works in the same way as the `lag` function, except that it returns a subsequent value as determined by the offset. Changing the *ORDER BY* from ascending (*ASC*) to descending (*DESC*) in a time series has the effect of turning a `lag` statement into the equivalent of a `lead` statement. Alternatively, a negative integer can be used as the *OFFSET* value to return a value from a subsequent row.

Let's apply this to our retail sales data set to calculate MoM and YoY growth. In this section, we'll focus on book store sales, since I'm a real book store nerd. First, we'll develop our intuition about what is returned by the `lag` function by returning both the lagging month and the lagging sales values:

```
SELECT kind_of_business, sales_month, sales
,lag(sales_month) over (partition by kind_of_business
                        order by sales_month
                        ) as prev_month
,lag(sales) over (partition by kind_of_business
                  order by sales_month
                  ) as prev_month_sales
FROM retail_sales
WHERE kind_of_business = 'Book stores'
```

```
;

kind_of_business  sales_month  sales  prev_month  prev_month_sales
----------------  -----------  -----  ----------  ----------------
Book stores       1992-01-01   790    (null)      (null)
Book stores       1992-02-01   539    1992-01-01  790
Book stores       1992-03-01   535    1992-02-01  539
...               ...          ...    ...         ...
```

For each row, the previous `sales_month` is returned, as well as the `sales` for that month, and we can confirm this by inspecting the first few lines of the result set. The first row has null for `prev_month` and `prev_month_sales` since there is no earlier record in this data set. With an understanding of the values returned by the `lag` function, we can calculate the percent change from the previous value:

```
SELECT kind_of_business, sales_month, sales
,(sales / lag(sales) over (partition by kind_of_business
                           order by sales_month)
 - 1) * 100 as pct_growth_from_previous
FROM retail_sales
WHERE kind_of_business = 'Book stores'
;

kind_of_business  sales_month  sales  pct_growth_from_previous
----------------  -----------  -----  ------------------------
Book stores       1992-01-01   790    (null)
Book stores       1992-02-01   539    -31.77
Book stores       1992-03-01   535    -0.74
...               ...          ...    ...
```

Sales dropped 31.8% from January to February, due at least in part to the seasonal decline after the holidays and the return to school for the spring semester. Sales were down only 0.7% from February to March.

The calculation for the YoY comparison is similar, but first we need to aggregate sales to the yearly level. Since we're looking at only one `kind_of_business`, I'll drop that field from the rest of the examples to simplify the code:

```
SELECT sales_year, yearly_sales
,lag(yearly_sales) over (order by sales_year) as prev_year_sales
,(yearly_sales / lag(yearly_sales) over (order by sales_year)
 -1) * 100 as pct_growth_from_previous
FROM
(
    SELECT date_part('year',sales_month) as sales_year
    ,sum(sales) as yearly_sales
    FROM retail_sales
    WHERE kind_of_business = 'Book stores'
    GROUP BY 1
) a
;
```

```
sales_year  yearly_sales  prev_year_sales  pct_growth_from_previous
----------  ------------  ---------------  ------------------------
1992.0      8327          (null)           (null)
1993.0      9108          8327             9.37
1994.0      10107         9108             10.96
...         ...           ...              ...
```

Sales grew more than 9.3% from 1992 to 1993, and almost 11% from 1993 to 1994. These period-over-period calculations are useful, but they don't quite allow us to analyze the seasonality in the data set. For example, in Figure 3-17 the MoM percent growth values are plotted, and they contain just as much seasonality as the original time series.

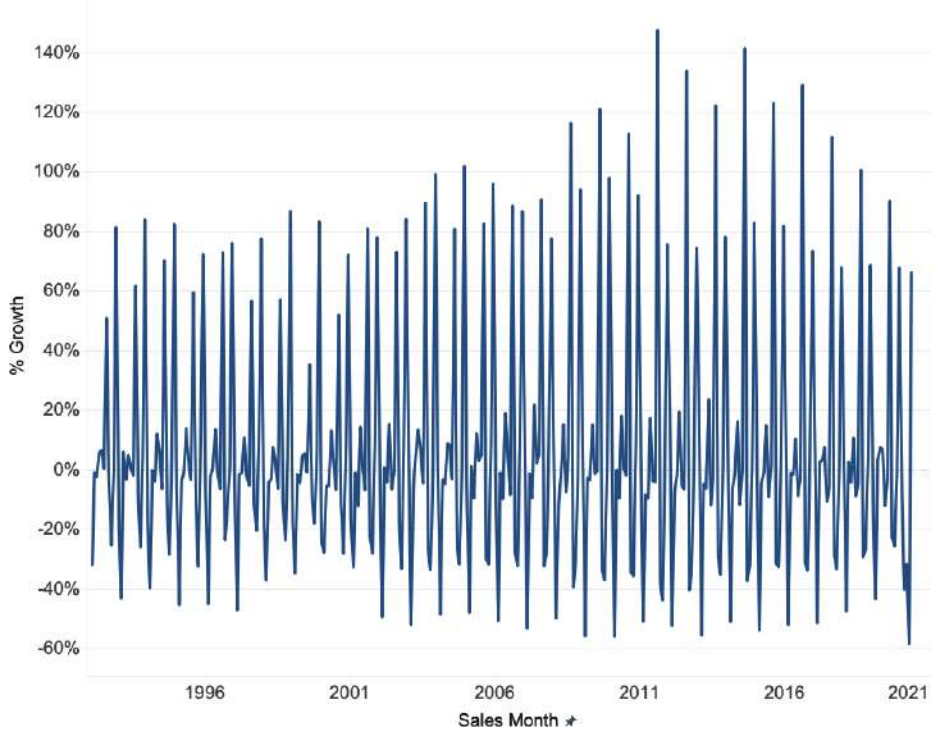


Figure 3-17. Percent growth from previous month for US retail book store sales

To tackle this, the next section will demonstrate how to use SQL to compare current values to the values for the same month in the previous year.

Period-over-Period Comparisons: Same Month Versus Last Year

Comparing data for one time period to data for a similar previous time period can be a useful way to control for seasonality. The previous time period may be the same day of the week in the previous week, the same month in the previous year, or another variation that makes sense for the data set.

To accomplish this comparison, we can use the `lag` function along with some clever partitioning: the unit of time with which we want to compare the current value. In this case, we will compare monthly `sales` to the `sales` for the same month in the previous year. For example, January `sales` will be compared to prior year January `sales`, February `sales` will be compared to prior year February `sales`, and so on.

First, recall that the `date_part` function returns a numeric value when used with the "month" argument:

```
SELECT sales_month
,date_part('month',sales_month)
FROM retail_sales
WHERE kind_of_business = 'Book stores'
;

sales_month  date_part
-----------  ---------
1992-01-01   1.0
1992-02-01   2.0
1992-03-01   3.0
...          ...
```

Next, we include the `date_part` in the *PARTITION BY* clause so that the window function looks up the value for the matching month number from the prior year.

This is an example of how window function clauses can include calculations in addition to database fields, giving them even more versatility. I find it useful to check intermediate results to build intuition about what the final query will return, so first we'll confirm that the `lag` function with `partition by date_part('month', sales_month)` returns the intended values:

```
SELECT sales_month, sales
,lag(sales_month) over (partition by date_part('month',sales_month)
                        order by sales_month
                        ) as prev_year_month
,lag(sales) over (partition by date_part('month',sales_month)
                  order by sales_month
                  ) as prev_year_sales
FROM retail_sales
WHERE kind_of_business = 'Book stores'
;
```

```
sales_month  sales  prev_year_month  prev_year_sales
-----------  -----  ---------------  ---------------
1992-01-01   790    (null)           (null)
1993-01-01   998    1992-01-01       790
1994-01-01   1053   1993-01-01       998
...          ...    ...              ...
1992-02-01   539    (null)           (null)
1993-02-01   568    1992-02-01       539
1994-02-01   635    1993-02-01       568
...          ...     ...             ...
```

The first `lag` function returns the same month for the prior year, which we can verify by looking at the `prev_year_month` value. The row for the 1993-01-01 `sales_month` returns 1992-01-01 for the `prev_year_month` as intended, and the `prev_year_sales` of 790 match the `sales` we can see in the 1992-01-01 row. Notice that the `prev_year_month` and `prev_year_sales` are null for 1992 since there are no prior records in the data set.

Now that we're confident the `lag` function as written returns the correct values, we can calculate comparison metrics such as absolute difference and percent change from previous:

```
SELECT sales_month, sales
,sales - lag(sales) over (partition by date_part('month',sales_month)
                          order by sales_month
                          ) as absolute_diff
,(sales / lag(sales) over (partition by date_part('month',sales_month)
                          order by sales_month)
 - 1) * 100 as pct_diff
FROM retail_sales
WHERE kind_of_business = 'Book stores'
;

sales_month  sales  absolute_diff  pct_diff
-----------  -----  -------------  --------
1992-01-01   790    (null)         (null)
1993-01-01   998    208            26.32
1994-01-01   1053   55             5.51
...          ...    ...            ...
```

We can now graph the results in Figure 3-18 and more easily see the months where growth was unusually high, such as January 2002, or unusually low, such as December 2001.

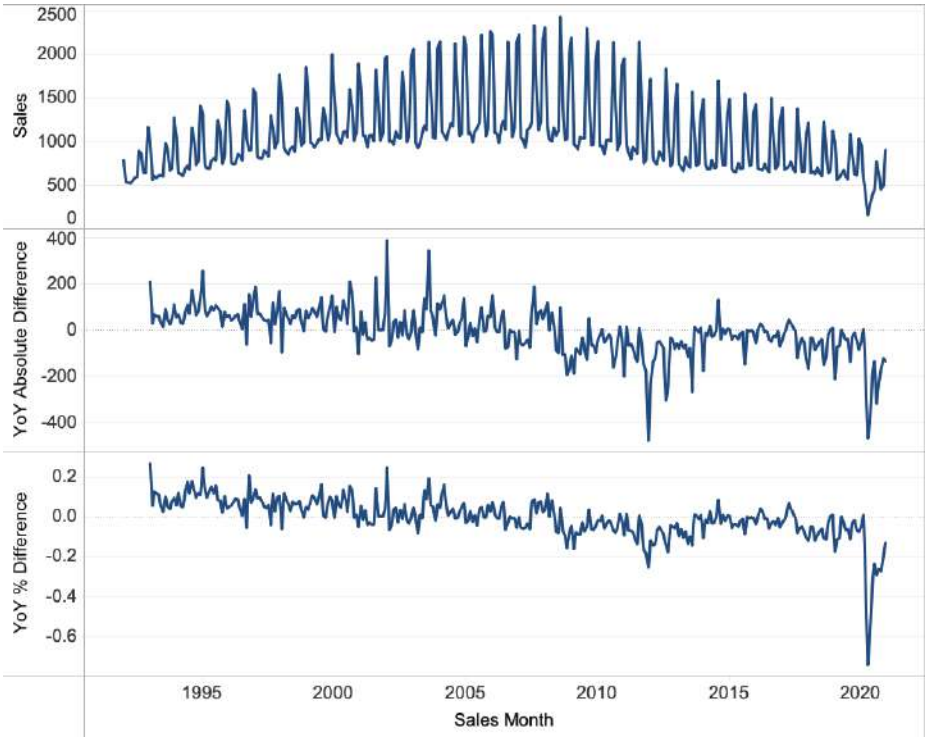


Figure 3-18. Book store sales, YoY absolute difference in sales, and YoY percent growth

Another useful analysis tool is to create a graph that lines up the same time period—in this case, months—with a line for each time series—in this case, years. To do this, we'll create a result set that has a row for each month number or name, and a column for each of the years we want to consider. To get the month, we can use either the `date_part` or the `to_char` function, depending on whether we want numeric or text values for the months. Then we'll pivot the data using an aggregate function.

This example uses the `max` aggregate, but depending on the analysis, a `sum`, `count`, or other aggregation might be appropriate. We'll zoom in on 1992 through 1994 for this example:

```
SELECT date_part('month',sales_month) as month_number
,to_char(sales_month,'Month') as month_name
,max(case when date_part('year',sales_month) = 1992 then sales end)
 as sales_1992
,max(case when date_part('year',sales_month) = 1993 then sales end)
 as sales_1993
,max(case when date_part('year',sales_month) = 1994 then sales end)
 as sales_1994
FROM retail_sales
```

```
WHERE kind_of_business = 'Book stores'
 and sales_month between '1992-01-01' and '1994-12-01'
GROUP BY 1,2
;

month_number  month_name  sales_1992  sales_1993  sales_1994
------------  ----------  ----------  ----------  ----------
1.0           January     790         998         1053
2.0           February    539         568         635
3.0           March       535         602         634
4.0           April       523         583         610
5.0           May         552         612         684
6.0           June        589         618         724
7.0           July        592         607         678
8.0           August      894         983         1154
9.0           September   861         903         1022
10.0          October     645         669         732
11.0          November    642         692         772
12.0          December    1165        1273        1409
```

By lining the data up in this way, we can see some trends immediately. December sales are the highest monthly sales of the year. Sales in 1994 were higher every month than sales in 1992 and 1993. The August-to-September sales bump is visible, and particularly easy to spot in 1994.

With a graph of the data, as in Figure 3-19, the trends are much easier to identify. Sales increased year to year in every month, though the increases were larger in some months than others. With this data and graph in hand, we can start to construct a story about book store sales that might help with inventory planning or scheduling of marketing promotions or might serve as a piece of evidence in a wider story about US retail sales.

With SQL there are a number of techniques for cutting through the noise of seasonality to compare data in time series. In this section, we've seen how to compare current values to prior comparable periods using `lag` functions and how to pivot the data with `date_part`, `to_char`, and aggregate functions. Next, I'll show some techniques for comparing multiple prior periods in order to further control for noisy time series data.

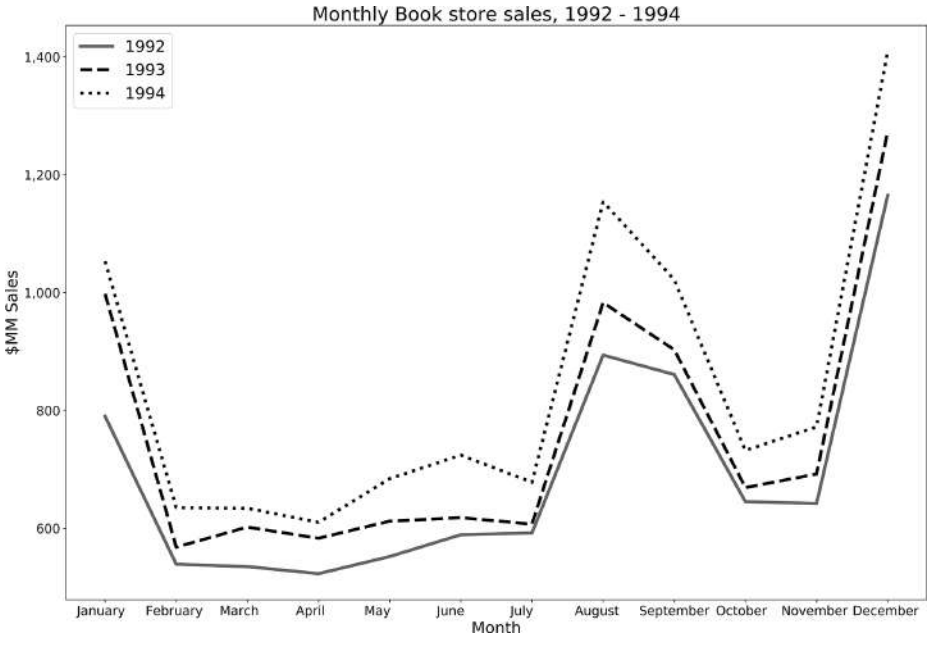


Figure 3-19. Book store sales for 1992–1994, aligned by month

Comparing to Multiple Prior Periods

Comparing data to prior comparable periods is a useful way to reduce the noise that arises from seasonality. Sometimes comparing to a single prior period is insufficient, particularly if that prior period was impacted by unusual events. Comparing a Monday to the previous Monday is difficult if one of them was a holiday. The month in the prior year might be unusual due to economic events, severe weather, or a site outage that changed typical behavior. Comparing current values to an aggregate of multiple prior periods can help smooth out these fluctuations. These techniques also combine what we've learned about using SQL to calculate rolling time periods and comparable prior period results.

The first technique uses the `lag` function, as in the last section, but here we'll take advantage of the optional offset value. Recall that when no offset is provided to `lag`, the function returns the immediate prior value according to the *PARTITION BY* and *ORDER BY* clauses. An offset value of 2 skips over the immediate prior value and returns the value prior to that, an offset value of 3 returns the value from 3 rows back, and so on.

For this example, we'll compare the current month's sales to the same month's sales over three prior years. As usual, first we'll inspect the returned values to confirm the SQL is working as expected:

```
SELECT sales_month, sales
,lag(sales,1) over (partition by date_part('month',sales_month)
                    order by sales_month
                    ) as prev_sales_1
,lag(sales,2) over (partition by date_part('month',sales_month)
                    order by sales_month
                    ) as prev_sales_2
,lag(sales,3) over (partition by date_part('month',sales_month)
                    order by sales_month
                    ) as prev_sales_3
FROM retail_sales
WHERE kind_of_business = 'Book stores'
;

sales_month  sales  prev_sales_1  prev_sales_2  prev_sales_3
-----------  -----  ------------  ------------  ------------
1992-01-01   790    (null)        (null)        (null)
1993-01-01   998    790           (null)        (null)
1994-01-01   1053   998           790           (null)
1995-01-01   1308   1053          998           790
1996-01-01   1373   1308          1053          998
...          ...    ...           ...           ...
```

Null is returned where no prior record exists, and we can confirm that the correct same month, prior year value appears. From here we can calculate whatever comparison metric the analysis calls for—in this case, the percent of the rolling average of three prior periods:

```
SELECT sales_month, sales
,sales / ((prev_sales_1 + prev_sales_2 + prev_sales_3) / 3)
 as pct_of_3_prev
FROM
(
    SELECT sales_month, sales
    ,lag(sales,1) over (partition by date_part('month',sales_month)
                        order by sales_month
                        ) as prev_sales_1
    ,lag(sales,2) over (partition by date_part('month',sales_month)
                        order by sales_month
                        ) as prev_sales_2
    ,lag(sales,3) over (partition by date_part('month',sales_month)
                        order by sales_month
                        ) as prev_sales_3
    FROM retail_sales
    WHERE kind_of_business = 'Book stores'
) a
;

sales_month  sales  pct_of_3_prev
-----------  -----  -------------
1995-01-01   1308   138.12
1996-01-01   1373   122.69
```

```
1997-01-01   1558   125.24
...          ...    ...
2017-01-01   1386   94.67
2018-01-01   1217   84.98
2019-01-01   1004   74.75
...          ...    ...
```

We can see from the result that book sales grew from the prior three-year rolling average in the mid-1990s, but the picture was different in the late 2010s, when sales were a shrinking percentage of that three-year rolling average each year.

You might have noticed that this problem resembles one we saw earlier when calculating rolling time windows. As an alternative to the last example, we can use an `avg` window function with a frame clause. To accomplish this, the *PARTITION BY* will use the same `date_part` function, and the *ORDER BY* is the same. A frame clause is added to include `rows between 3 preceding and 1 preceding`. This includes the values in the 1, 2, and 3 rows prior but excludes the value in the current row:

```
SELECT sales_month, sales
,sales / avg(sales) over (partition by date_part('month',sales_month)
                          order by sales_month
                          rows between 3 preceding and 1 preceding
                          ) as pct_of_prev_3
FROM retail_sales
WHERE kind_of_business = 'Book stores'
;

sales_month  sales  pct_of_prev_3
-----------  -----  -------------
1995-01-01   1308   138.12
1996-01-01   1373   122.62
1997-01-01   1558   125.17
...          ...    ...
2017-01-01   1386   94.62
2018-01-01   1217   84.94
2019-01-01   1004   74.73
...          ...    ...
```

The results match those of the previous example, confirming that the alternative code is equivalent.

If you look closely, you'll notice that the decimal place values are slightly different in the result using the three `lag` windows and the result using the `avg` window function. This is due to how the database handles decimal rounding in intermediate calculations. For many analyses, the difference won't matter, but pay careful attention if you're working with financial or other highly regulated data.

Analyzing data with seasonality often involves trying to reduce noise in order to make clear conclusions about the underlying trends in the data. Comparing data points against multiple prior time periods can give us an even smoother trend to compare to and determine what is actually happening in the current time period. This does require that the data include enough history to make these comparisons, but when we have a long enough time series, it can be insightful.

Conclusion

Time series analysis is a powerful way to analyze data sets. We've seen how to set up our data for analysis with date and time manipulations. We talked about date dimensions and saw how to apply them to calculating rolling time windows. We looked at period-over-period calculations and how to analyze data with seasonality patterns. In the next chapter, we'll delve deep into a related topic that extends on time series analysis: cohort analysis.

About the Author

Cathy Tanimura has a passion for connecting people and organizations to the data they need to make an impact. She has been analyzing data for over 20 years across a wide range of industries, from finance to B2B software to consumer services. She has experience analyzing data with SQL across most of the major proprietary and open source databases. She has built and managed data teams and data infrastructure at a number of leading tech companies. Cathy is also a frequent speaker at top conferences, on topics including building data cultures, data-driven product development, and inclusive data analysis.

Colophon

The animal on the cover of *SQL for Data Analysis* is a green magpie (*Cissa chinensis*). Usually referred to as the common green magpie, this jewel-toned bird is a member of the crow family. Found throughout the lowland evergreen and bamboo forests of northeastern India, central Thailand, Malaysia, Sumatra, and northwestern Borneo, this species of bird is noisy and highly social. In the wild, they can be identified by their jade-colored plumage, which contrasts elegantly with their red beak and a black band running along the eyes. They also have a white-tipped tail and reddish wings.

Highly social and noisy, the green magpie can be identified by its piercing shrieks followed by a hollow and decisive-sounding "chup" note. They are also often difficult to spot because they glide from tree to tree in the middle-upper levels of the forest. They build their nests in trees, large shrubs, and tangles of various climbing vines. Sometimes referred to as the hunting cissas, they are primarily carnivorous—consuming a variety of invertebrates, as well as young birds and eggs, small reptiles, and mammals.

The green magpie is fascinating because of its ability to change colors. Although they are jade green in the wild, they have been observed to turn distinctly turquoise in captivity. They get their green coloration from a combination of two sources: a special feather structure that produces blue coloring due to the feather refracting light, and carotenoids—yellow, orange, and red pigments that come from the bird's diet. Prolonged exposure to harsh sunlight destroys the carotenoids, hence making the bird appear turquoise.

The green magpie species has an extremely large range and although the population trend seems to be decreasing, the decline is not rapid enough to push the species into the Vulnerable category. As such, their current conservation status is "Least Concern."

The cover illustration is by Karen Montgomery, based on a black-and-white engraving from *English Cyclopedia*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.

O'REILLY®

Database Internals

A Deep Dive into How Distributed Data Systems Work

Free Chapters

compliments of

SingleStore

Alex Petrov

SingleStore

The Single Database

For Your Data-Intensive Applications

Switch to the unified database with 10 to 100x the performance at one-third the cost of legacy infrastructures. SingleStore delivers unmatched speed, scale, and agility in one powerful, cloud-native relational database.

Pure Speed
Fast transactions, fast analytics

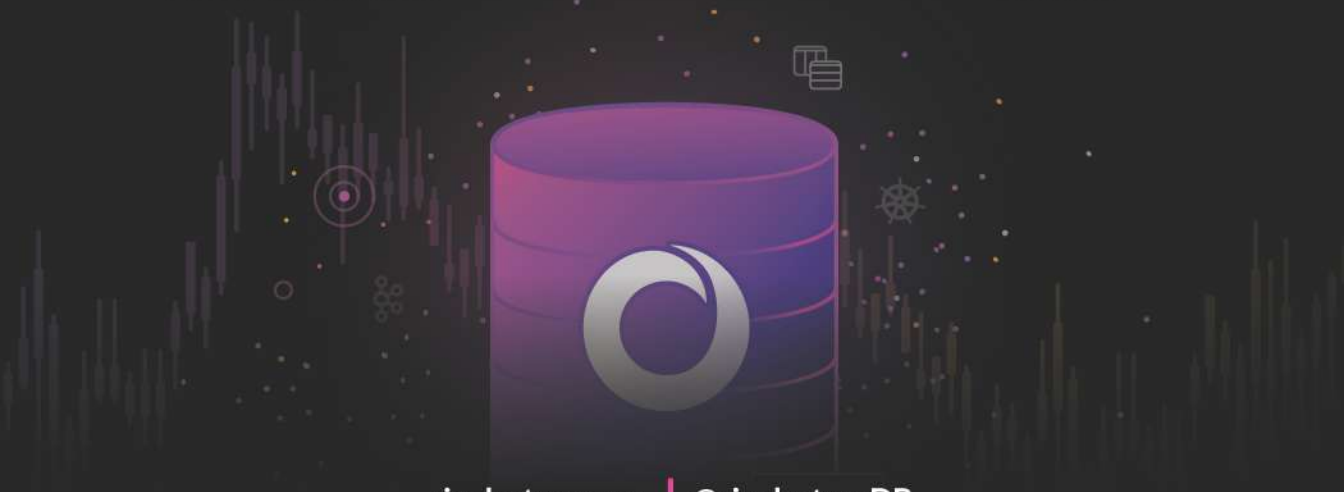
Universal Storage
Patented single table type for transactions & analytics

Fast Ingestion
Pipelines - Load data with updates

Multi-Model
Geospatial, Time-Series, Semi-Structured, & more

Run Anywhere
Your cloud or mine?

Compatibility
ANSI SQL & MySQL Ecosystem



singlestore.com | @singlestoreDB

Database Internals

A Deep Dive into How Distributed Data Systems Work

This excerpt contains Chapters 1, 8, and 13. The complete book is available on the O'Reilly Online Learning Platform and through other retailers.

Alex Petrov

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Database Internals
by Alex Petrov

Printed in Canada.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Mike Loukides
Development Editor: Michele Cronin
Production Editor: Christopher Faucher
Copyeditor: Kim Cofer
Proofreader: Sonia Saruba

Indexer: Judith McConville
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

October 2019: First Edition

Revision History for the First Edition

2019-09-12: First Release
2019-10-25: Second Release
2020-01-31: Third Release
2020-09-04: Fourth Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492040347* for release details.

978-1-492-04034-7

[MBP]

Table of Contents

PART I
Storage Engines

The primary job of any database management system is reliably storing data and making it available for users. We use databases as a primary source of data, helping us to share it between the different parts of our applications. Instead of finding a way to store and retrieve information and inventing a new way to organize data every time we create a new app, we use databases. This way we can concentrate on application logic instead of infrastructure.

Since the term *database management system* (DBMS) is quite bulky, throughout this book we use more compact terms, *database system* and *database*, to refer to the same concept.

Databases are modular systems and consist of multiple parts: a transport layer accepting requests, a query processor determining the most efficient way to run queries, an execution engine carrying out the operations, and a storage engine (see "DBMS Architecture" on page 8).

The *storage engine* (or database engine) is a software component of a database management system responsible for storing, retrieving, and managing data in memory and on disk, designed to capture a persistent, long-term memory of each node [REED78]. While databases can respond to complex queries, storage engines look at the data more granularly and offer a simple data manipulation API, allowing users to create, update, delete, and retrieve records. One way to look at this is that database management systems are applications built on top of storage engines, offering a schema, a query language, indexing, transactions, and many other useful features.

For flexibility, both keys and values can be arbitrary sequences of bytes with no prescribed form. Their sorting and representation semantics are defined in higher-level subsystems. For example, you can use `int32` (32-bit integer) as a key in one of the tables, and `ascii` (ASCII string) in the other; from the storage engine perspective both keys are just serialized entries.

Storage engines such as BerkeleyDB, LevelDB and its descendant RocksDB, LMDB and its descendant libmdbx, Sophia, HaloDB, and many others were developed independently from the database management systems they're now embedded into. Using pluggable storage engines has enabled database developers to bootstrap database systems using existing storage engines, and concentrate on the other subsystems.

At the same time, clear separation between database system components opens up an opportunity to switch between different engines, potentially better suited for particular use cases. For example, MySQL, a popular database management system, has several storage engines, including InnoDB, MyISAM, and RocksDB (in the MyRocks distribution). MongoDB allows switching between WiredTiger, In-Memory, and the (now-deprecated) MMAPv1 storage engines.

Comparing Databases

Your choice of database system may have long-term consequences. If there's a chance that a database is not a good fit because of performance problems, consistency issues, or operational challenges, it is better to find out about it earlier in the development cycle, since it can be nontrivial to migrate to a different system. In some cases, it may require substantial changes in the application code.

Every database system has strengths and weaknesses. To reduce the risk of an expensive migration, you can invest some time before you decide on a specific database to build confidence in its ability to meet your application's needs.

Trying to compare databases based on their components (e.g., which storage engine they use, how the data is shared, replicated, and distributed, etc.), their rank (an arbitrary popularity value assigned by consultancy agencies such as ThoughtWorks or database comparison websites such as DB-Engines or Database of Databases), or implementation language (C++, Java, or Go, etc.) can lead to invalid and premature conclusions. These methods can be used only for a high-level comparison and can be as coarse as choosing between HBase and SQLite, so even a superficial understanding of how each database works and what's inside it can help you land a more weighted conclusion.

Every comparison should start by clearly defining the goal, because even the slightest bias may completely invalidate the entire investigation. If you're searching for a database that would be a good fit for the workloads you have (or are planning to facilitate), the best thing you can do is to simulate these workloads against different

database systems, measure the performance metrics that are important for you, and compare results. Some issues, especially when it comes to performance and scalability, start showing only after some time or as the capacity grows. To detect potential problems, it is best to have long-running tests in an environment that simulates the real-world production setup as closely as possible.

Simulating real-world workloads not only helps you understand how the database performs, but also helps you learn how to operate, debug, and find out how friendly and helpful its community is. Database choice is always a combination of these factors, and performance often turns out *not* to be the most important aspect: it's usually much better to use a database that slowly saves the data than one that quickly loses it.

To compare databases, it's helpful to understand the use case in great detail and define the current and anticipated variables, such as:

- Schema and record sizes
- Number of clients
- Types of queries and access patterns
- Rates of the read and write queries
- Expected changes in any of these variables

Knowing these variables can help to answer the following questions:

- Does the database support the required queries?
- Is this database able to handle the amount of data we're planning to store?
- How many read and write operations can a single node handle?
- How many nodes should the system have?
- How do we expand the cluster given the expected growth rate?
- What is the maintenance process?

Having these questions answered, you can construct a test cluster and simulate your workloads. Most databases already have stress tools that can be used to reconstruct specific use cases. If there's no standard stress tool to generate realistic randomized workloads in the database ecosystem, it might be a red flag. If something prevents you from using default tools, you can try one of the existing general-purpose tools, or implement one from scratch.

If the tests show positive results, it may be helpful to familiarize yourself with the database code. Looking at the code, it is often useful to first understand the parts of the database, how to find the code for different components, and then navigate through those. Having even a rough idea about the database codebase helps you bet-

ter understand the log records it produces, its configuration parameters, and helps you find issues in the application that uses it and even in the database code itself.

It'd be great if we could use databases as black boxes and never have to take a look inside them, but the practice shows that sooner or later a bug, an outage, a performance regression, or some other problem pops up, and it's better to be prepared for it. If you know and understand database internals, you can reduce business risks and improve chances for a quick recovery.

One of the popular tools used for benchmarking, performance evaluation, and comparison is Yahoo! Cloud Serving Benchmark (YCSB). YCSB offers a framework and a common set of workloads that can be applied to different data stores. Just like anything generic, this tool should be used with caution, since it's easy to make wrong conclusions. To make a fair comparison and make an educated decision, it is necessary to invest enough time to understand the real-world conditions under which the database has to perform, and tailor benchmarks accordingly.

TPC-C Benchmark

The Transaction Processing Performance Council (TPC) has a set of benchmarks that database vendors use for comparing and advertising performance of their products. TPC-C is an online transaction processing (OLTP) benchmark, a mixture of read-only and update transactions that simulate common application workloads.

This benchmark concerns itself with the performance and correctness of executed concurrent transactions. The main performance indicator is *throughput*: the number of transactions the database system is able to process per minute. Executed transactions are required to preserve ACID properties and conform to the set of properties defined by the benchmark itself.

This benchmark does not concentrate on any particular business segment, but provides an abstract set of actions important for most of the applications for which OLTP databases are suitable. It includes several tables and entities such as warehouses, stock (inventory), customers and orders, specifying table layouts, details of transactions that can be performed against these tables, the minimum number of rows per table, and data durability constraints.

This doesn't mean that benchmarks can be used *only* to compare databases. Benchmarks can be useful to define and test details of the service-level agreement,[1] under-

1 The service-level agreement (or SLA) is a commitment by the service provider about the quality of provided services. Among other things, the SLA can include information about latency, throughput, jitter, and the number and frequency of failures.

standing system requirements, capacity planning, and more. The more knowledge you have about the database before using it, the more time you'll save when running it in production.

Choosing a database is a long-term decision, and it's best to keep track of newly released versions, understand what exactly has changed and why, and have an upgrade strategy. New releases usually contain improvements and fixes for bugs and security issues, but may introduce new bugs, performance regressions, or unexpected behavior, so testing new versions before rolling them out is also critical. Checking how database implementers were handling upgrades previously might give you a good idea about what to expect in the future. Past smooth upgrades do not guarantee that future ones will be as smooth, but complicated upgrades in the past might be a sign that future ones won't be easy, either.

Understanding Trade-Offs

As users, we can see how databases behave under different conditions, but when working on databases, we have to make choices that influence this behavior directly.

Designing a storage engine is definitely more complicated than just implementing a textbook data structure: there are many details and edge cases that are hard to get right from the start. We need to design the physical data layout and organize pointers, decide on the serialization format, understand how data is going to be garbage-collected, how the storage engine fits into the semantics of the database system as a whole, figure out how to make it work in a concurrent environment, and, finally, make sure we never lose any data, under any circumstances.

Not only there are many things to decide upon, but most of these decisions involve trade-offs. For example, if we save records in the order they were inserted into the database, we can store them quicker, but if we retrieve them in their lexicographical order, we have to re-sort them before returning results to the client. As you will see throughout this book, there are many different approaches to storage engine design, and every implementation has its own upsides and downsides.

When looking at different storage engines, we discuss their benefits and shortcomings. If there was an absolutely optimal storage engine for every conceivable use case, everyone would just use it. But since it does not exist, we need to choose wisely, based on the workloads and use cases we're trying to facilitate.

There are many storage engines, using all sorts of data structures, implemented in different languages, ranging from low-level ones, such as C, to high-level ones, such as Java. All storage engines face the same challenges and constraints. To draw a parallel with city planning, it is possible to build a city for a specific population and choose to build *up* or build *out*. In both cases, the same number of people will fit into the city, but these approaches lead to radically different lifestyles. When building the city up,

people live in apartments and population density is likely to lead to more traffic in a smaller area; in a more spread-out city, people are more likely to live in houses, but commuting will require covering larger distances.

Similarly, design decisions made by storage engine developers make them better suited for different things: some are optimized for low read or write latency, some try to maximize density (the amount of stored data per node), and some concentrate on operational simplicity.

You can find complete algorithms that can be used for the implementation and other additional references in the chapter summaries. Reading this book should make you well equipped to work productively with these sources and give you a solid understanding of the existing alternatives to concepts described there.

CHAPTER 1
Introduction and Overview

Database management systems can serve different purposes: some are used primarily for temporary *hot* data, some serve as a long-lived *cold* storage, some allow complex analytical queries, some only allow accessing values by the key, some are optimized to store time-series data, and some store large blobs efficiently. To understand differences and draw distinctions, we start with a short classification and overview, as this helps us to understand the scope of further discussions.

Terminology can sometimes be ambiguous and hard to understand without a complete context. For example, distinctions between *column* and *wide column* stores that have little or nothing to do with each other, or how *clustered* and *nonclustered indexes* relate to *index-organized tables*. This chapter aims to disambiguate these terms and find their precise definitions.

We start with an overview of database management system architecture (see "DBMS Architecture" on page 8), and discuss system components and their responsibilities. After that, we discuss the distinctions among the database management systems in terms of a storage medium (see "Memory- Versus Disk-Based DBMS" on page 10), and layout (see "Column- Versus Row-Oriented DBMS" on page 12).

These two groups do not present a full taxonomy of database management systems and there are many other ways they're classified. For example, some sources group DBMSs into three major categories:

Online transaction processing (OLTP) databases
: These handle a large number of user-facing requests and transactions. Queries are often predefined and short-lived.

Online analytical processing (OLAP) databases
: These handle complex aggregations. OLAP databases are often used for analytics and data warehousing, and are capable of handling complex, long-running ad hoc queries.

Hybrid transactional and analytical processing (HTAP)
: These databases combine properties of both OLTP and OLAP stores.

There are many other terms and classifications: key-value stores, relational databases, document-oriented stores, and graph databases. These concepts are not defined here, since the reader is assumed to have a high-level knowledge and understanding of their functionality. Because the concepts we discuss here are widely applicable and are used in most of the mentioned types of stores in some capacity, complete taxonomy is not necessary or important for further discussion.

Since Part I of this book focuses on the storage and indexing structures, we need to understand the high-level data organization approaches, and the relationship between the data and index files (see "Data Files and Index Files" on page 17).

Finally, in "Buffering, Immutability, and Ordering" on page 21, we discuss three techniques widely used to develop efficient storage structures and how applying these techniques influences their design and implementation.

DBMS Architecture

There's no common blueprint for database management system design. Every database is built slightly differently, and component boundaries are somewhat hard to see and define. Even if these boundaries exist on paper (e.g., in project documentation), in code seemingly independent components may be coupled because of performance optimizations, handling edge cases, or architectural decisions.

Sources that describe database management system architecture (for example, HELLERSTEIN07, WEIKUM01, ELMASRI11, and GARCIAMOLINA08), define components and relationships between them differently. The architecture presented in Figure 1-1 demonstrates some of the common themes in these representations.

Database management systems use a *client/server model*, where database system instances (*nodes*) take the role of servers, and application instances take the role of clients.

Client requests arrive through the *transport* subsystem. Requests come in the form of queries, most often expressed in some query language. The transport subsystem is also responsible for communication with other nodes in the database cluster.

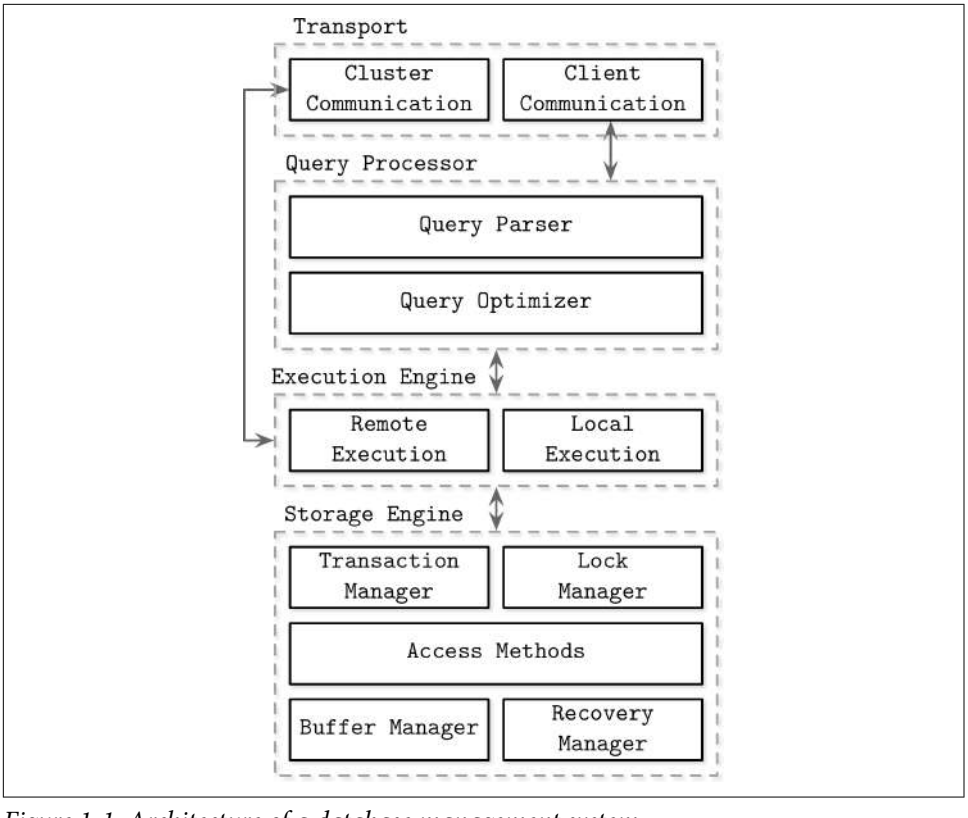


Figure 1-1. Architecture of a database management system

Upon receipt, the transport subsystem hands the query over to a *query processor*, which parses, interprets, and validates it. Later, access control checks are performed, as they can be done fully only after the query is interpreted.

The parsed query is passed to the *query optimizer*, which first eliminates impossible and redundant parts of the query, and then attempts to find the most efficient way to execute it based on internal statistics (index cardinality, approximate intersection size, etc.) and data placement (which nodes in the cluster hold the data and the costs associated with its transfer). The optimizer handles both relational operations required for query resolution, usually presented as a dependency tree, and optimizations, such as index ordering, cardinality estimation, and choosing access methods.

The query is usually presented in the form of an *execution plan* (or *query plan*): a sequence of operations that have to be carried out for its results to be considered complete. Since the same query can be satisfied using different execution plans that can vary in efficiency, the optimizer picks the best available plan.

The execution plan is carried out by the *execution engine*, which aggregates the results of local and remote operations. *Remote execution* can involve writing and reading data to and from other nodes in the cluster, and replication.

Local queries (coming directly from clients or from other nodes) are executed by the *storage engine*. The storage engine has several components with dedicated responsibilities:

Transaction manager
: This manager schedules transactions and ensures they cannot leave the database in a logically inconsistent state.

Lock manager
: This manager locks on the database objects for the running transactions, ensuring that concurrent operations do not violate physical data integrity.

Access methods (storage structures)
: These manage access and organizing data on disk. Access methods include heap files and storage structures such as B-Trees (see Chapter 2) or LSM Trees (see Chapter 7).

Buffer manager
: This manager caches data pages in memory (see Chapter 5).

Recovery manager
: This manager maintains the operation log and restoring the system state in case of a failure (see Chapter 5).

Together, transaction and lock managers are responsible for concurrency control (see Chapter 5): they guarantee logical and physical data integrity while ensuring that concurrent operations are executed as efficiently as possible.

Memory- Versus Disk-Based DBMS

Database systems store data in memory and on disk. *In-memory database management systems* (sometimes called *main memory DBMS*) store data *primarily* in memory and use the disk for recovery and logging. *Disk-based* DBMS hold *most* of the data on disk and use memory for caching disk contents or as a temporary storage. Both types of systems use the disk to a certain extent, but main memory databases store their contents almost exclusively in RAM.

Accessing memory has been and remains several orders of magnitude faster than accessing disk,[1] so it is compelling to use memory as the primary storage, and it becomes more economically feasible to do so as memory prices go down. However, RAM prices still remain high compared to persistent storage devices such as SSDs and HDDs.

Main memory database systems are different from their disk-based counterparts not only in terms of a primary storage medium, but also in which data structures, organization, and optimization techniques they use.

Databases using memory as a primary data store do this mainly because of performance, comparatively low access costs, and access granularity. Programming for main memory is also significantly simpler than doing so for the disk. Operating systems abstract memory management and allow us to think in terms of allocating and freeing arbitrarily sized memory chunks. On disk, we have to manage data references, serialization formats, freed memory, and fragmentation manually.

The main limiting factors on the growth of in-memory databases are RAM volatility (in other words, lack of durability) and costs. Since RAM contents are not persistent, software errors, crashes, hardware failures, and power outages can result in data loss. There are ways to ensure durability, such as uninterrupted power supplies and battery-backed RAM, but they require additional hardware resources and operational expertise. In practice, it all comes down to the fact that disks are easier to maintain and have significantly lower prices.

The situation is likely to change as the availability and popularity of Non-Volatile Memory (NVM) technologies grow. NVM storage reduces or completely eliminates (depending on the exact technology) asymmetry between read and write latencies, further improves read and write performance, and allows byte-addressable access.

Durability in Memory-Based Stores

In-memory database systems maintain backups on disk to provide durability and prevent loss of the volatile data. Some databases store data exclusively in memory, without any durability guarantees, but we do not discuss them in the scope of this book.

Before the operation can be considered complete, its results have to be written to a sequential log file. We discuss write-ahead logs in more detail in Chapter 5. To avoid replaying complete log contents during startup or after a crash, in-memory stores maintain a *backup copy*. The backup copy is maintained as a sorted disk-based structure, and modifications to this structure are often asynchronous (decoupled from cli-

1 You can find a visualization and comparison of disk, memory access latencies, and many other relevant numbers over the years at *https://people.eecs.berkeley.edu/~rcs/research/interactive_latency.html*.

ent requests) and applied in batches to reduce the number of I/O operations. During recovery, database contents can be restored from the backup and logs.

Log records are usually applied to backup in batches. After the batch of log records is processed, backup holds a database *snapshot* for a specific point in time, and log contents up to this point can be discarded. This process is called *checkpointing*. It reduces recovery times by keeping the disk-resident database most up-to-date with log entries without requiring clients to block until the backup is updated.

It is unfair to say that the in-memory database is the equivalent of an on-disk database with a huge page cache (see Chapter 5). Even though pages are *cached* in memory, serialization format and data layout incur additional overhead and do not permit the same degree of optimization that in-memory stores can achieve.

Disk-based databases use specialized storage structures, optimized for disk access. In memory, pointers can be followed comparatively quickly, and random memory access is significantly faster than the random disk access. Disk-based storage structures often have a form of wide and short trees (see Chapter 2), while memory-based implementations can choose from a larger pool of data structures and perform optimizations that would otherwise be impossible or difficult to implement on disk [GARCIAMOLINA92]. Similarly, handling variable-size data on disk requires special attention, while in memory it's often a matter of referencing the value with a pointer.

For some use cases, it is reasonable to assume that an entire dataset is going to fit in memory. Some datasets are bounded by their real-world representations, such as student records for schools, customer records for corporations, or inventory in an online store. Each record takes up not more than a few Kb, and their number is limited.

Column- Versus Row-Oriented DBMS

Most database systems store a set of *data records*, consisting of *columns* and *rows* in *tables*. *Field* is an intersection of a column and a row: a single value of some type. Fields belonging to the same column usually have the same data type. For example, if we define a table holding user records, all names would be of the same type and belong to the same column. A collection of values that belong logically to the same record (usually identified by the key) constitutes a row.

One of the ways to classify databases is by how the data is stored on disk: row- or column-wise. Tables can be partitioned either horizontally (storing values belonging to the same row together), or vertically (storing values belonging to the same column together). Figure 1-2 depicts this distinction: (a) shows the values partitioned column-wise, and (b) shows the values partitioned row-wise.

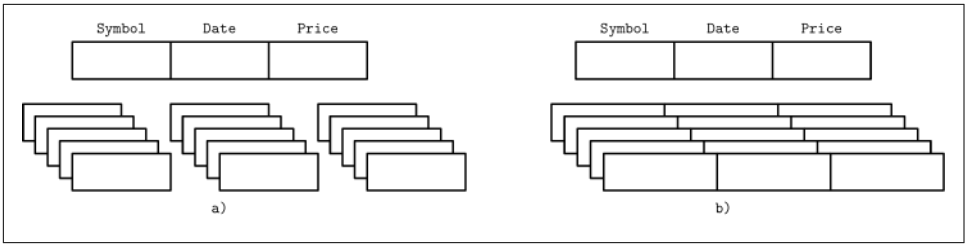


Figure 1-2. Data layout in column- and row-oriented stores

Examples of row-oriented database management systems are abundant: MySQL, PostgreSQL, and most of the traditional relational databases. The two pioneer open source column-oriented stores are MonetDB and C-Store (C-Store is an open source predecessor to Vertica).

Row-Oriented Data Layout

Row-oriented database management systems store data in records or *rows*. Their layout is quite close to the tabular data representation, where every row has the same set of fields. For example, a row-oriented database can efficiently store user entries, holding names, birth dates, and phone numbers:

```
| ID | Name  | Birth Date  | Phone Number   |
| 10 | John  | 01 Aug 1981 | +1 111 222 333 |
| 20 | Sam   | 14 Sep 1988 | +1 555 888 999 |
| 30 | Keith | 07 Jan 1984 | +1 333 444 555 |
```

This approach works well for cases where several fields constitute the record (name, birth date, and a phone number) uniquely identified by the key (in this example, a monotonically incremented number). All fields representing a single user record are often read together. When creating records (for example, when the user fills out a registration form), we write them together as well. At the same time, each field can be modified individually.

Since row-oriented stores are most useful in scenarios when we have to access data by row, storing entire rows together improves spatial locality[2] [DENNING68].

Because data on a persistent medium such as a disk is typically accessed block-wise (in other words, a minimal unit of disk access is a block), a single block will contain data for all columns. This is great for cases when we'd like to access an entire user record, but makes queries accessing individual fields of multiple user records (for

2 Spatial locality is one of the Principles of Locality, stating that if a memory location is accessed, its nearby memory locations will be accessed in the near future.

example, queries fetching only the phone numbers) more expensive, since data for the other fields will be paged in as well.

Column-Oriented Data Layout

Column-oriented database management systems partition data *vertically* (i.e., by column) instead of storing it in rows. Here, values for the same column are stored contiguously on disk (as opposed to storing rows contiguously as in the previous example). For example, if we store historical stock market prices, price quotes are stored together. Storing values for different columns in separate files or file segments allows efficient queries by column, since they can be read in one pass rather than consuming entire rows and discarding data for columns that weren't queried.

Column-oriented stores are a good fit for analytical workloads that compute aggregates, such as finding trends, computing average values, etc. Processing complex aggregates can be used in cases when logical records have multiple fields, but some of them (in this case, price quotes) have different importance and are often consumed together.

From a logical perspective, the data representing stock market price quotes can still be expressed as a table:

```
| ID | Symbol | Date        | Price     |
| 1  | DOW    | 08 Aug 2018 | 24,314.65 |
| 2  | DOW    | 09 Aug 2018 | 24,136.16 |
| 3  | S&P    | 08 Aug 2018 | 2,414.45  |
| 4  | S&P    | 09 Aug 2018 | 2,232.32  |
```

However, the physical column-based database layout looks entirely different. Values belonging to the same column are stored closely together:

```
Symbol: 1:DOW; 2:DOW; 3:S&P; 4:S&P
Date:   1:08 Aug 2018; 2:09 Aug 2018; 3:08 Aug 2018; 4:09 Aug 2018
Price:  1:24,314.65; 2:24,136.16; 3:2,414.45; 4:2,232.32
```

To reconstruct data tuples, which might be useful for joins, filtering, and multirow aggregates, we need to preserve some metadata on the column level to identify which data points from other columns it is associated with. If you do this explicitly, each value will have to hold a key, which introduces duplication and increases the amount of stored data. Some column stores use implicit identifiers (*virtual IDs*) instead and use the position of the value (in other words, its offset) to map it back to the related values [ABADI13].

During the last several years, likely due to a rising demand to run complex analytical queries over growing datasets, we've seen many new column-oriented file formats such as Apache Parquet, Apache ORC, RCFile, as well as column-oriented stores, such as Apache Kudu, ClickHouse, and many others [ROY12].

Distinctions and Optimizations

It is not sufficient to say that distinctions between row and column stores are only in the way the data is stored. Choosing the data layout is just one of the steps in a series of possible optimizations that columnar stores are targeting.

Reading multiple values for the same column in one run significantly improves cache utilization and computational efficiency. On modern CPUs, vectorized instructions can be used to process multiple data points with a single CPU instruction[3] [DREPPER07].

Storing values that have the same data type together (e.g., numbers with other numbers, strings with other strings) offers a better compression ratio. We can use different compression algorithms depending on the data type and pick the most effective compression method for each case.

To decide whether to use a column- or a row-oriented store, you need to understand your *access patterns*. If the read data is consumed in records (i.e., most or all of the columns are requested) and the workload consists mostly of point queries and range scans, the row-oriented approach is likely to yield better results. If scans span many rows, or compute aggregate over a subset of columns, it is worth considering a column-oriented approach.

Wide Column Stores

Column-oriented databases should not be mixed up with *wide column stores*, such as BigTable or HBase, where data is represented as a multidimensional map, columns are grouped into *column families* (usually storing data of the same type), and inside each column family, data is stored row-wise. This layout is best for storing data retrieved by a key or a sequence of keys.

A canonical example from the Bigtable paper [CHANG06] is a Webtable. A Webtable stores snapshots of web page contents, their attributes, and the relations among them at a specific timestamp. Pages are identified by the reversed URL, and all attributes (such as page *content* and *anchors*, representing links between pages) are identified by the timestamps at which these snapshots were taken. In a simplified way, it can be represented as a nested map, as Figure 1-3 shows.

3 Vectorized instructions, or Single Instruction Multiple Data (SIMD), describes a class of CPU instructions that perform the same operation on multiple data points.

```
{
    "com.cnn.www": {
        contents: {
            t6: html: "<html>..."
            t5: html: "<html>..."
            t3: html: "<html>..."
        }
        anchor: {
            t9: cnnsi.com: "CNN"
            t8: my.look.ca: "CNN.com"
        }
    }
    "com.example.www": {
        contents: {
            t5: html: "<html>..."
        }
        anchor: {}
    }
}
```

Figure 1-3. Conceptual structure of a Webtable

Data is stored in a multidimensional sorted map with hierarchical indexes: we can locate the data related to a specific web page by its reversed URL and its contents or anchors by the timestamp. Each row is indexed by its *row key*. Related columns are grouped together in *column families*—`contents` and `anchor` in this example—which are stored on disk separately. Each column inside a column family is identified by the *column key*, which is a combination of the column family name and a qualifier (`html`, `cnnsi.com`, `my.look.ca` in this example). Column families store multiple versions of data by timestamp. This layout allows us to quickly locate the higher-level entries (web pages, in this case) and their parameters (versions of content and links to the other pages).

While it is useful to understand the conceptual representation of wide column stores, their physical layout is somewhat different. A schematic representation of the data layout in column families is shown in Figure 1-4: column families are stored separately, but in each column family, the data belonging to the same key is stored together.

Column Family: contents

Row Key	Timestamp	Qualifier	Value
com.cnn.www	t3	html	"<html>..."
com.cnn.www	t5	html	"<html>..."
com.cnn.www	t6	html	"<html>..."
com.example.www	t5	html	"<html>..."

Column Family: anchor

Row Key	Timestamp	Qualifier	Value
com.cnn.www	t8	cnnsi.com	"CNN"
com.cnn.www	t5	my.look.ca	"CNN.com"

Figure 1-4. Physical structure of a Webtable

Data Files and Index Files

The primary goal of a database system is to store data and to allow quick access to it. But how is the data organized? Why do we need a database management system and not just a bunch of files? How does file organization improve efficiency?

Database systems do use files for storing the data, but instead of relying on filesystem hierarchies of directories and files for locating records, they compose files using implementation-specific formats. The main reasons to use specialized file organization over flat files are:

Storage efficiency
: Files are organized in a way that minimizes storage overhead per stored data record.

Access efficiency
: Records can be located in the smallest possible number of steps.

Update efficiency
: Record updates are performed in a way that minimizes the number of changes on disk.

Database systems store *data records*, consisting of multiple fields, in tables, where each table is usually represented as a separate file. Each record in the table can be looked up using a *search key*. To locate a record, database systems use *indexes*: auxiliary data structures that allow it to efficiently locate data records without scanning an entire table on every access. Indexes are built using a subset of fields identifying the record.

A database system usually separates *data files* and *index files*: data files store data records, while index files store record metadata and use it to locate records in data

files. Index files are typically smaller than the data files. Files are partitioned into *pages*, which typically have the size of a single or multiple disk blocks. Pages can be organized as sequences of records or as *slotted pages* (see Chapter 3).

New records (insertions) and updates to the existing records are represented by key/value pairs. Most modern storage systems *do not* delete data from pages explicitly. Instead, they use *deletion markers* (also called *tombstones*), which contain deletion metadata, such as a key and a timestamp. Space occupied by the records *shadowed* by their updates or deletion markers is reclaimed during garbage collection, which reads the pages, writes the live (i.e., nonshadowed) records to the new place, and discards the shadowed ones.

Data Files

Data files (sometimes called *primary files*) can be implemented as *index-organized tables* (IOT), *heap-organized tables* (heap files), or *hash-organized tables* (hashed files).

Records in heap files are not required to follow any particular order, and most of the time they are placed in a write order. This way, no additional work or file reorganization is required when new pages are appended. Heap files require additional index structures, pointing to the locations where data records are stored, to make them searchable.

In hashed files, records are stored in buckets, and the hash value of the key determines which bucket a record belongs to. Records in the bucket can be stored in append order or sorted by key to improve lookup speed.

Index-organized tables (IOTs) store data records in the index itself. Since records are stored in key order, range scans in IOTs can be implemented by sequentially scanning its contents.

Storing data records in the index allows us to reduce the number of disk seeks by at least one, since after traversing the index and locating the searched key, we do not have to address a separate file to find the associated data record.

When records are stored in a separate file, index files hold *data entries*, uniquely identifying data records and containing enough information to locate them in the data file. For example, we can store file *offsets* (sometimes called *row locators*), locations of data records in the data file, or bucket IDs in the case of hash files. In index-organized tables, data entries hold actual data records.

Index Files

An index is a structure that organizes data records on disk in a way that facilitates efficient retrieval operations. Index files are organized as specialized structures that

map keys to locations in data files where the records identified by these keys (in the case of heap files) or primary keys (in the case of index-organized tables) are stored.

An index on a *primary* (data) file is called the *primary index*. In most cases we can also assume that the primary index is built over a primary key or a set of keys identified as primary. All other indexes are called *secondary*.

Secondary indexes can point directly to the data record, or simply store its primary key. A pointer to a data record can hold an offset to a heap file or an index-organized table. Multiple secondary indexes can point to the same record, allowing a single data record to be identified by different fields and located through different indexes. While primary index files hold a unique entry per search key, secondary indexes may hold several entries per search key [GARCIAMOLINA08].

If the order of data records follows the search key order, this index is called *clustered* (also known as clustering). Data records in the clustered case are usually stored in the same file or in a *clustered file*, where the key order is preserved. If the data is stored in a separate file, and its order does not follow the key order, the index is called *nonclustered* (sometimes called unclustered).

Figure 1-5 shows the difference between the two approaches:

- a) An index-organized table, where data records are stored directly in the index file.
- b) An index file storing the offsets and a separate file storing data records.

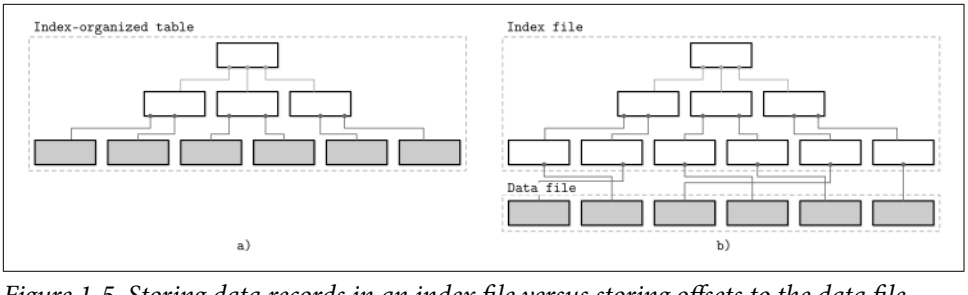


Figure 1-5. Storing data records in an index file versus storing offsets to the data file (index segments shown in white; segments holding data records shown in gray)

Index-organized tables store information in index order and are clustered by definition. Primary indexes are *most often* clustered. Secondary indexes are nonclustered by definition, since they're used to facilitate access by keys other than the primary one. Clustered indexes can be both index-organized or have separate index and data files.

Many database systems have an inherent and explicit *primary key*, a set of columns that uniquely identify the database record. In cases when the primary key is not specified, the storage engine can create an *implicit* primary key (for example, MySQL InnoDB adds a new auto-increment column and fills in its values automatically).

This terminology is used in different kinds of database systems: relational database systems (such as MySQL and PostgreSQL), Dynamo-based NoSQL stores (such as Apache Cassandra and in Riak), and document stores (such as MongoDB). There can be some project-specific naming, but most often there's a clear mapping to this terminology.

Primary Index as an Indirection

There are different opinions in the database community on whether data records should be referenced directly (through file offset) or via the primary key index.[4]

Both approaches have their pros and cons and are better discussed in the scope of a complete implementation. By referencing data directly, we can reduce the number of disk seeks, but have to pay a cost of updating the pointers whenever the record is updated or relocated during a maintenance process. Using indirection in the form of a primary index allows us to reduce the cost of pointer updates, but has a higher cost on a read path.

Updating just a couple of indexes might work if the workload mostly consists of reads, but this approach does not work well for write-heavy workloads with multiple indexes. To reduce the costs of pointer updates, instead of payload offsets, some implementations use primary keys for indirection. For example, MySQL InnoDB uses a primary index and performs two lookups: one in the secondary index, and one in a primary index when performing a query [TARIQ11]. This adds an overhead of a primary index lookup instead of following the offset directly from the secondary index.

Figure 1-6 shows how the two approaches are different:

- a) Two indexes reference data entries directly from secondary index files.
- b) A secondary index goes through the indirection layer of a primary index to locate the data entries.

4 The original post that has stirred up the discussion was controversial and one-sided, but you can refer to the presentation comparing MySQL and PostgreSQL index and storage formats, which references the original source as well.

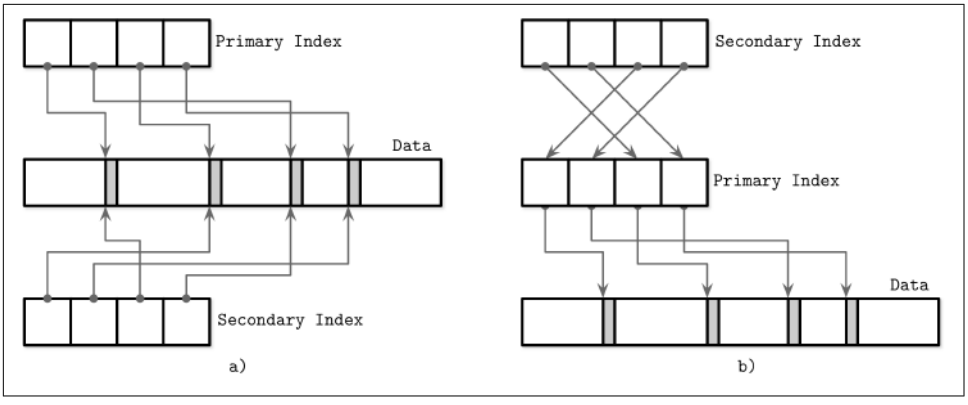


Figure 1-6. Referencing data tuples directly (a) versus using a primary index as indirection (b)

It is also possible to use a hybrid approach and store both data file offsets and primary keys. First, you check if the data offset is still valid and pay the extra cost of going through the primary key index if it has changed, updating the index file after finding a new offset.

Buffering, Immutability, and Ordering

A storage engine is based on some data structure. However, these structures do not describe the semantics of caching, recovery, transactionality, and other things that storage engines add on top of them.

In the next chapters, we will start the discussion with B-Trees (see Chapter 2) and try to understand why there are so many B-Tree variants, and why new database storage structures keep emerging.

Storage structures have three common variables: they use *buffering* (or avoid using it), use *immutable* (or mutable) files, and store values *in order* (or out of order). Most of the distinctions and optimizations in storage structures discussed in this book are related to one of these three concepts.

Buffering
: This defines whether or not the storage structure chooses to collect a certain amount of data in memory before putting it on disk. Of course, every on-disk structure has to use buffering to *some* degree, since the smallest unit of data transfer to and from the disk is a *block*, and it is desirable to write full blocks. Here, we're talking about avoidable buffering, something storage engine implementers *choose* to do. One of the first optimizations we discuss in this book is adding in-memory buffers to B-Tree nodes to amortize I/O costs (see Chapter 6). However, this is not the only way we can apply buffering. For example, two-

component LSM Trees (see Chapter 7), despite their similarities with B-Trees, use buffering in an entirely different way, and combine buffering with immutability.

Mutability (or immutability)
: This defines whether or not the storage structure reads parts of the file, updates them, and writes the updated results at the same location in the file. Immutable structures are *append-only*: once written, file contents are not modified. Instead, modifications are appended to the end of the file. There are other ways to implement immutability. One of them is *copy-on-write* (see Chapter 6), where the modified page, holding the updated version of the record, is written to the *new* location in the file, instead of its original location. Often the distinction between LSM and B-Trees is drawn as immutable against in-place update storage, but there are structures (for example, Chapter 6) that are inspired by B-Trees but are immutable.

Ordering
: This is defined as whether or not the *data records* are stored in the key order in the pages on disk. In other words, the keys that sort closely are stored in contiguous segments on disk. Ordering often defines whether or not we can efficiently scan the *range* of records, not only locate the individual data records. Storing data out of order (most often, in insertion order) opens up for some write-time optimizations. For example, Bitcask and WiscKey (see Chapter 7) store data records directly in append-only files.

Of course, a brief discussion of these three concepts is not enough to show their power, and we'll continue this discussion throughout the rest of the book.

Summary

In this chapter, we've discussed the architecture of a database management system and covered its primary components.

To highlight the importance of disk-based structures and their difference from in-memory ones, we discussed memory- and disk-based stores. We came to the conclusion that disk-based structures are important for both types of stores, but are used for different purposes.

To understand how access patterns influence database system design, we discussed column- and row-oriented database management systems and the primary factors that set them apart from each other. To start a conversation about *how the data is stored*, we covered data and index files.

Lastly, we introduced three core concepts: buffering, immutability, and ordering. We will use them throughout this book to highlight properties of the storage engines that use them.

Further Reading

If you'd like to learn more about the concepts mentioned in this chapter, you can refer to the following sources:

Database architecture

Hellerstein, Joseph M., Michael Stonebraker, and James Hamilton. 2007. "Architecture of a Database System." *Foundations and Trends in Databases* 1, no. 2 (February): 141-259. *https://doi.org/10.1561/1900000002*.

Column-oriented DBMS

Abadi, Daniel, Peter Boncz, Stavros Harizopoulos, Stratos Idreaos, and Samuel Madden. 2013. *The Design and Implementation of Modern Column-Oriented Database Systems*. Hanover, MA: Now Publishers Inc.

In-memory DBMS

Faerber, Frans, Alfons Kemper, and Per-Åke Alfons. 2017. *Main Memory Database Systems*. Hanover, MA: Now Publishers Inc.

PART II
Distributed Systems

> A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.
>
> —Leslie Lamport

Without distributed systems, we wouldn't be able to make phone calls, transfer money, or exchange information over long distances. We use distributed systems daily. Sometimes, even without acknowledging it: any client/server application is a distributed system.

For many modern software systems, *vertical* scaling (scaling by running the same software on a bigger, faster machine with more CPU, RAM, or faster disks) isn't viable. Bigger machines are more expensive, harder to replace, and may require special maintenance. An alternative is to scale *horizontally*: to run software on multiple machines connected over the network and working as a single logical entity.

Distributed systems might differ both in size, from a handful to hundreds of machines, and in characteristics of their participants, from small handheld or sensor devices to high-performance computers.

The time when database systems were mainly running on a single node is long gone, and most modern database systems have multiple nodes connected in clusters to increase storage capacity, improve performance, and enhance availability.

Even though some of the theoretical breakthroughs in distributed computing aren't new, most of their practical application happened relatively recently. Today, we see increasing interest in the subject, more research, and new development being done.

Basic definitions

In a distributed system, we have several *participants* (sometimes called *processes*, *nodes*, or *replicas*). Each participant has its own local *state*. Participants communicate by exchanging *messages* using communication *links* between them.

Processes can access the time using a *clock*, which can be *logical* or *physical*. Logical clocks are implemented using a kind of monotonically growing counter. Physical clocks, also called *wall clocks*, are bound to a notion of time in the physical world and are accessible through process-local means; for example, through an operating system.

It's impossible to talk about distributed systems without mentioning the inherent difficulties caused by the fact that its parts are located apart from each other. Remote processes communicate through links that can be slow and unreliable, which makes knowing the exact state of the remote process more complicated.

Most of the research in the distributed systems field is related to the fact that nothing is entirely reliable: communication channels may delay, reorder, or fail to deliver the messages; processes may pause, slow down, crash, go out of control, or suddenly stop responding.

There are many themes in common in the fields of concurrent and distributed programming, since CPUs are tiny distributed systems with links, processors, and communication protocols. You'll see many parallels with concurrent programming in Chapter 11. However, most of the primitives can't be reused directly because of the costs of communication between remote parties, and the unreliability of links and processes.

To overcome the difficulties of the distributed environment, we need to use a particular class of algorithms, *distributed algorithms*, which have notions of local and remote state and execution and work despite unreliable networks and component failures. We describe algorithms in terms of *state* and *steps* (or *phases*), with *transitions* between them. Each process executes the algorithm steps locally, and a combination of local executions and process interactions constitutes a distributed algorithm.

Distributed algorithms describe the local behavior and interaction of multiple independent nodes. Nodes communicate by sending messages to each other. Algorithms define participant roles, exchanged messages, states, transitions, executed steps, properties of the delivery medium, timing assumptions, failure models, and other characteristics that describe processes and their interactions.

Distributed algorithms serve many different purposes:

Coordination
: A process that supervises the actions and behavior of several workers.

Cooperation
: Multiple participants relying on one another for finishing their tasks.

Dissemination
: Processes cooperating in spreading the information to all interested parties quickly and reliably.

Consensus
: Achieving agreement among multiple processes.

In this book, we talk about algorithms in the context of their usage and prefer a practical approach over purely academic material. First, we cover all necessary abstractions, the processes and the connections between them, and progress to building more complex communication patterns. We start with UDP, where the sender doesn't have any guarantees on whether or not its message has reached its destination; and finally, to achieve consensus, where multiple processes agree on a specific value.

CHAPTER 8
Introduction and Overview

What makes distributed systems inherently different from single-node systems? Let's take a look at a simple example and try to see. In a single-threaded program, we define variables and the execution process (a set of steps).

For example, we can define a variable and perform simple arithmetic operations over it:

```
int x = 1;
x += 2;
x *= 2;
```

We have a single execution history: we declare a variable, increment it by two, then multiply it by two, and get the result: `6`. Let's say that, instead of having one execution thread performing these operations, we have two threads that have read and write access to variable `x`.

Concurrent Execution

As soon as two execution threads are allowed to access the variable, the exact outcome of the concurrent step execution is unpredictable, unless the steps are synchronized between the threads. Instead of a single possible outcome, we end up with four, as Figure 8-1 shows.[1]

1 Interleaving, where the multiplier reads before the adder, is left out for brevity, since it yields the same result as a).

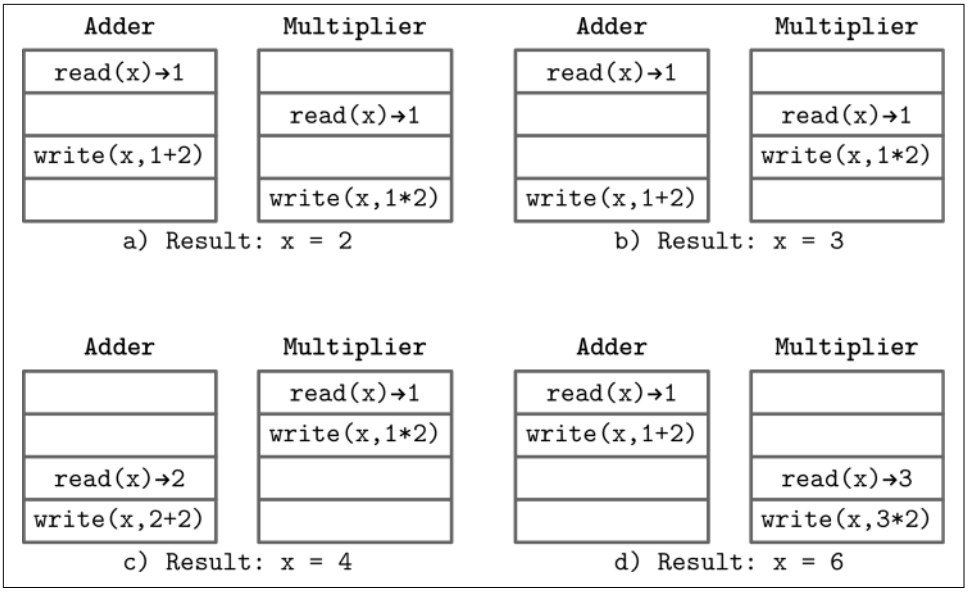


Figure 8-1. Possible interleavings of concurrent executions

- a) x = 2, if both threads read an initial value, the adder writes its value, but it is overwritten with the multiplication result.
- b) x = 3, if both threads read an initial value, the multiplier writes its value, but it is overwritten with the addition result.
- c) x = 4, if the multiplier can read the initial value and execute its operation before the adder starts.
- d) x = 6, if the adder can read the initial value and execute its operation before the multiplier starts.

Even before we can cross a single node boundary, we encounter the first problem in distributed systems: *concurrency*. Every concurrent program has some properties of a distributed system. Threads access the shared state, perform some operations locally, and propagate the results back to the shared variables.

To define execution histories precisely and reduce the number of possible outcomes, we need *consistency models*. Consistency models describe concurrent executions and establish an order in which operations can be executed and made visible to the participants. Using different consistency models, we can constraint or relax the number of states the system can be in.

There is a lot of overlap in terminology and research in the areas of distributed systems and concurrent computing, but there are also some differences. In a concurrent system, we can have *shared memory*, which processors can use to exchange the

information. In a distributed system, each processor has its local state and participants communicate by passing messages.

Concurrent and Parallel

We often use the terms *concurrent* and *parallel* computing interchangeably, but these concepts have a slight semantic difference. When two sequences of steps execute concurrently, both of them are in progress, but only one of them is executed at any moment. If two sequences execute in parallel, their steps can be executed simultaneously. Concurrent operations overlap in time, while parallel operations are executed by multiple processors [WEIKUM01].

Joe Armstrong, creator of the Erlang programming language, gave an example: concurrent execution is like having two queues to a single coffee machine, while parallel execution is like having two queues to two coffee machines. That said, the vast majority of sources use the term concurrency to describe systems with several parallel execution threads, and the term parallelism is rarely used.

Shared State in a Distributed System

We can try to introduce some notion of shared memory to a distributed system, for example, a single source of information, such as database. Even if we solve the problems with concurrent access to it, we still cannot guarantee that all processes are in sync.

To access this database, processes have to go over the communication medium by sending and receiving messages to query or modify the state. However, what happens if one of the processes does not receive a response from the database for a longer time? To answer this question, we first have to define what *longer* even means. To do this, the system has to be described in terms of *synchrony*: whether the communication is fully asynchronous, or whether there are some timing assumptions. These timing assumptions allow us to introduce operation timeouts and retries.

We do not know whether the database hasn't responded because it's overloaded, unavailable, or slow, or because of some problems with the network on the way to it. This describes a *nature* of a crash: processes may crash by failing to participate in further algorithm steps, having a temporary failure, or by omitting some of the messages. We need to define a *failure model* and describe ways in which failures can occur before we decide how to treat them.

A property that describes system reliability and whether or not it can continue operating correctly in the presence of failures is called *fault tolerance*. Failures are inevitable, so we need to build systems with reliable components, and eliminating a single point of failure in the form of the aforementioned single-node database can be the

first step in this direction. We can do this by introducing some *redundancy* and adding a backup database. However, now we face a different problem: how do we keep *multiple copies* of shared state in sync?

So far, trying to introduce shared state to our simple system has left us with more questions than answers. We now know that sharing state is not as simple as just introducing a database, and have to take a more granular approach and describe interactions in terms of independent processes and passing messages between them.

Fallacies of Distributed Computing

In an ideal case, when two computers talk over the network, everything works just fine: a process opens up a connection, sends the data, gets responses, and everyone is happy. Assuming that operations always succeed and nothing can go wrong is dangerous, since when something does break and our assumptions turn out to be wrong, systems behave in ways that are hard or impossible to predict.

Most of the time, assuming that the *network is reliable* is a reasonable thing to do. It has to be reliable to at least some extent to be useful. We've all been in the situation when we tried to establish a connection to the remote server and got a `Network is Unreachable` error instead. But even if it is possible to establish a connection, a successful *initial* connection to the server does not guarantee that the link is stable, and the connection can get interrupted at any time. The message might've reached the remote party, but the response could've gotten lost, or the connection was interrupted before the response was delivered.

Network switches break, cables get disconnected, and network configurations can change at any time. We should build our system by handling all of these scenarios gracefully.

A connection can be stable, but we can't expect remote calls to be as fast as the local ones. We should make as few assumptions about latency as possible and never assume that *latency is zero*. For our message to reach a remote server, it has to go through several software layers, and a physical medium such as optic fiber or a cable. All of these operations are not instantaneous.

Michael Lewis, in his *Flash Boys* book (Simon and Schuster), tells a story about companies spending millions of dollars to reduce latency by several milliseconds to able to access stock exchanges faster than the competition. This is a great example of using latency as a competitive advantage, but it's worth mentioning that, according to some other studies, such as BARTLETT16, the chance of stale-quote arbitrage (the ability to profit from being able to know prices and execute orders faster than the competition) doesn't give fast traders the ability to exploit markets.

Learning our lessons, we've added retries, reconnects, and removed the assumptions about instantaneous execution, but this still turns out not to be enough. When increasing the number, rates, and sizes of exchanged messages, or adding new processes to the existing network, we should not assume that *bandwidth is infinite*.

In 1994, Peter Deutsch published a now-famous list of assertions, titled "Fallacies of distributed computing," describing the aspects of distributed computing that are easy to overlook. In addition to network reliability, latency, and bandwidth assumptions, he describes some other problems. For example, network security, the possible presence of adversarial parties, intentional and unintentional topology changes that can break our assumptions about presence and location of specific resources, transport costs in terms of both time and resources, and, finally, the existence of a single authority having knowledge and control over the entire network.

Deutsch's list of distributed computing fallacies is pretty exhaustive, but it focuses on what can go wrong when we send messages from one process to another through the link. These concerns are valid and describe the most general and low-level complications, but unfortunately, there are many other assumptions we make about the distributed systems while designing and implementing them that can cause problems when operating them.

Processing

Before a remote process can send a response to the message it just received, it needs to perform some work locally, so we cannot assume that *processing is instantaneous*. Taking network latency into consideration is not enough, as operations performed by the remote processes aren't immediate, either.

Moreover, there's no guarantee that processing starts as soon as the message is delivered. The message may land in the pending queue on the remote server, and will have to wait there until all the messages that arrived before it are processed.

Nodes can be located closer or further from one another, have different CPUs, amounts of RAM, different disks, or be running different software versions and configurations. We cannot expect them to process requests at the same rate. If we have to wait for several remote servers working in parallel to respond to complete the task, the execution as a whole is as slow as the slowest remote server.

Contrary to the widespread belief, *queue capacity is not infinite* and piling up more requests won't do the system any good. *Backpressure* is a strategy that allows us to cope with producers that publish messages at a rate that is faster than the rate at which consumers can process them by slowing down the producers. Backpressure is

one of the least appreciated and applied concepts in distributed systems, often built post hoc instead of being an integral part of the system design.

Even though increasing the queue capacity might sound like a good idea and can help to pipeline, parallelize, and effectively schedule requests, nothing is happening to the messages while they're sitting in the queue and waiting for their turn. Increasing the queue size may negatively impact latency, since changing it has no effect on the processing rate.

In general, process-local queues are used to achieve the following goals:

Decoupling
: Receipt and processing are separated in time and happen independently.

Pipelining
: Requests in different stages are processed by independent parts of the system. The subsystem responsible for receiving messages doesn't have to block until the previous message is fully processed.

Absorbing short-time bursts
: System load tends to vary, but request inter-arrival times are hidden from the component responsible for request processing. Overall system latency increases because of the time spent in the queue, but this is usually still better than responding with a failure and retrying the request.

Queue size is workload- and application-specific. For relatively stable workloads, we can size queues by measuring task processing times and the average time each task spends in the queue before it is processed, and making sure that latency remains within acceptable bounds while throughput increases. In this case, queue sizes are relatively small. For unpredictable workloads, when tasks get submitted in bursts, queues should be sized to account for bursts and high load as well.

The remote server can work through requests quickly, but it doesn't mean that we always get a positive response from it. It can respond with a failure: it couldn't make a write, the searched value was not present, or it could've hit a bug. In summary, even the most favorable scenario still requires some attention from our side.

Clocks and Time

> Time is an illusion. Lunchtime doubly so.
>
> —Ford Prefect, *The Hitchhiker's Guide to the Galaxy*

Assuming that clocks on remote machines run in sync can also be dangerous. Combined with *latency is zero* and *processing is instantaneous*, it leads to different idiosyncrasies, especially in time-series and real-time data processing. For example, when collecting and aggregating data from participants with a different perception of time,

you should understand time drifts between them and normalize times accordingly, rather than relying on the source timestamp. Unless you use specialized high-precision time sources, you should not rely on timestamps for synchronization or ordering. Of course this doesn't mean we cannot or should not rely on time at all: in the end, any synchronous system uses *local* clocks for timeouts.

It's essential to always account for the possible time differences between the processes and the time required for the messages to get delivered and processed. For example, Spanner (see "Distributed Transactions with Spanner" on page 64) uses a special time API that returns a timestamp and uncertainty bounds to impose a strict transaction order. Some failure-detection algorithms rely on a shared notion of time and a guarantee that the clock drift is always within allowed bounds for correctness [GUPTA01].

Besides the fact that clock synchronization in a distributed system is hard, the *current* time is constantly changing: you can request a current POSIX timestamp from the operating system, and request another *current* timestamp after executing several steps, and the two will be different. This is a rather obvious observation, but understanding both a source of time and which exact moment the timestamp captures is crucial.

Understanding whether the clock source is monotonic (i.e., that it won't ever go backward) and how much the scheduled time-related operations might drift can be helpful, too.

State Consistency

Most of the previous assumptions fall into the *almost always false* category, but there are some that are better described as *not always true*: when it's easy to take a mental shortcut and simplify the model by thinking of it a specific way, ignoring some tricky edge cases.

Distributed algorithms do not always guarantee strict state consistency. Some approaches have looser constraints and allow state divergence between replicas, and rely on *conflict resolution* (an ability to detect and resolve diverged states within the system) and *read-time data repair* (bringing replicas back in sync during reads in cases where they respond with different results). You can find more information about these concepts in Chapter 12. Assuming that the state is fully consistent across the nodes may lead to subtle bugs.

An eventually consistent distributed database system might have the logic to handle replica disagreement by querying a quorum of nodes during reads, but assume that the database schema and the view of the cluster are strongly consistent. Unless we enforce consistency of this information, relying on that assumption may have severe consequences.

For example, there was a bug in Apache Cassandra, caused by the fact that schema changes propagate to servers at different times. If you tried to read from the database while the schema was propagating, there was a chance of corruption, since one server encoded results assuming one schema and the other one decoded them using a different schema.

Another example is a bug caused by the divergent view of the ring: if one of the nodes assumes that the other node holds data records for a key, but this other node has a different view of the cluster, reading or writing the data can result in misplacing data records or getting an empty response while data records are in fact happily present on the other node.

It is better to think about the possible problems in advance, even if a complete solution is costly to implement. By understanding and handling these cases, you can embed safeguards or change the design in a way that makes the solution more natural.

Local and Remote Execution

Hiding complexity behind an API might be dangerous. For example, if you have an iterator over the local dataset, you can reasonably predict what's going on behind the scenes, even if the storage engine is unfamiliar. Understanding the process of iteration over the remote dataset is an entirely different problem: you need to understand consistency and delivery semantics, data reconciliation, paging, merges, concurrent access implications, and many other things.

Simply hiding both behind the same interface, however useful, might be misleading. Additional API parameters may be necessary for debugging, configuration, and observability. We should always keep in mind that *local and remote execution are not the same* [WALDO96].

The most apparent problem with hiding remote calls is latency: remote invocation is many times more costly than the local one, since it involves two-way network transport, serialization/deserialization, and many other steps. Interleaving local and blocking remote calls may lead to performance degradation and unintended side effects [VINOSKI08].

Need to Handle Failures

It's OK to start working on a system assuming that all nodes are up and functioning normally, but thinking this is the case all the time is dangerous. In a long-running system, nodes can be taken down for maintenance (which usually involves a graceful shutdown) or crash for various reasons: software problems, out-of-memory killer [KERRISK10], runtime bugs, hardware issues, etc. Processes do fail, and the best thing you can do is be prepared for failures and understand how to handle them.

If the remote server doesn't respond, we do not always know the exact reason for it. It could be caused by the crash, a network failure, the remote process, or the link to it being slow. Some distributed algorithms use *heartbeat protocols* and *failure detectors* to form a hypothesis about which participants are alive and reachable.

Network Partitions and Partial Failures

When two or more servers cannot communicate with each other, we call the situation *network partition*. In "Perspectives on the CAP Theorem" [GILBERT12], Seth Gilbert and Nancy Lynch draw a distinction between the case when two participants cannot communicate with each other and when several groups of participants are isolated from one another, cannot exchange messages, and proceed with the algorithm.

General unreliability of the network (packet loss, retransmission, latencies that are hard to predict) are *annoying but tolerable*, while network partitions can cause much more trouble, since independent groups can proceed with execution and produce conflicting results. Network links can also fail asymmetrically: messages can still be getting delivered from one process to the other one, but not vice versa.

To build a system that is robust in the presence of failure of one or multiple processes, we have to consider cases of *partial failures* [TANENBAUM06] and how the system can continue operating even though a part of it is unavailable or functioning incorrectly.

Failures are hard to detect and aren't always visible in the same way from different parts of the system. When designing highly available systems, one should always think about edge cases: what if we did replicate the data, but received no acknowledgments? Do we need to retry? Is the data still going to be available for reads on the nodes that have sent acknowledgments?

Murphy's Law[2] tells us that the failures do happen. Programming folklore adds that the failures will happen in the worst way possible, so our job as distributed systems engineers is to make sure we reduce the number of scenarios where things go wrong and prepare for failures in a way that contains the damage they can cause.

It's impossible to prevent all failures, but we can still build a resilient system that functions correctly in their presence. The best way to design for failures is to test for them. It's close to impossible to think through every possible failure scenario and predict the behaviors of multiple processes. Setting up testing harnesses that create partitions, simulate bit rot [GRAY05], increase latencies, diverge clocks, and magnify relative processing speeds is the best way to go about it. Real-world distributed system setups

2 Murphy's Law is an adage that can be summarized as "Anything that can go wrong, will go wrong," which was popularized and is often used as an idiom in popular culture.

can be quite adversarial, unfriendly, and "creative" (however, in a very hostile way), so the testing effort should attempt to cover as many scenarios as possible.

Over the last few years, we've seen a few open source projects that help to recreate different failure scenarios. Toxiproxy can help to simulate network problems: limit the bandwidth, introduce latency, timeouts, and more. Chaos Monkey takes a more radical approach and exposes engineers to production failures by randomly shutting down services. CharybdeFS helps to simulate filesystem and hardware errors and failures. You can use these tools to test your software and make sure it behaves correctly in the presence of these failures. CrashMonkey, a filesystem agnostic record-replay-and-test framework, helps test data and metadata consistency for persistent files.

When working with distributed systems, we have to take fault tolerance, resilience, possible failure scenarios, and edge cases seriously. Similar to "given enough eyeballs, all bugs are shallow," we can say that a large enough cluster will eventually hit every possible issue. At the same time, given enough testing, we will be able to eventually find every existing problem.

Cascading Failures

We cannot always wholly isolate failures: a process tipping over under a high load increases the load for the rest of cluster, making it even more probable for the other nodes to fail. *Cascading failures* can propagate from one part of the system to the other, increasing the scope of the problem.

Sometimes, cascading failures can even be initiated by perfectly good intentions. For example, a node was offline for a while and did not receive the most recent updates. After it comes back online, helpful peers would like to help it to catch up with recent happenings and start streaming the data it's missing over to it, exhausting network resources or causing the node to fail shortly after the startup.

To protect a system from propagating failures and treat failure scenarios gracefully, *circuit breakers* can be used. In electrical engineering, circuit breakers protect expensive and hard-to-replace parts from overload or short circuit by interrupting the current flow. In software development, circuit breakers monitor failures and allow fallback mechanisms that can protect the system by steering away from the failing service, giving it some time to recover, and handling failing calls gracefully.

When the connection to one of the servers fails or the server does not respond, the client starts a reconnection loop. By that point, an overloaded server already has a hard time catching up with new connection requests, and client-side retries in a tight loop don't help the situation. To avoid that, we can use a *backoff* strategy. Instead of retrying immediately, clients wait for some time. Backoff can help us to avoid amplifying problems by scheduling retries and increasing the time window between subsequent requests.

Backoff is used to increase time periods between requests from a single client. However, different clients using the same backoff strategy can produce substantial load as well. To prevent *different* clients from retrying all at once after the backoff period, we can introduce *jitter*. Jitter adds small random time periods to backoff and reduces the probability of clients waking up and retrying at the same time.

Hardware failures, bit rot, and software errors can result in corruption that can propagate through standard delivery mechanisms. For example, corrupted data records can get replicated to the other nodes if they are not validated. Without validation mechanisms in place, a system can propagate corrupted data to the other nodes, potentially overwriting noncorrupted data records. To avoid that, we should use checksumming and validation to verify the integrity of any content exchanged between the nodes.

Overload and hotspotting can be avoided by planning and coordinating execution. Instead of letting peers execute operation steps independently, we can use a coordinator that prepares an execution plan based on the available resources and predicts the load based on the past execution data available to it.

In summary, we should always consider cases in which failures in one part of the system can cause problems elsewhere. We should equip our systems with circuit breakers, backoff, validation, and coordination mechanisms. Handling small isolated problems is always more straightforward than trying to recover from a large outage.

We've just spent an entire section discussing problems and potential failure scenarios in distributed systems, but we should see this as a warning and not as something that should scare us away.

Understanding what can go wrong, and carefully designing and testing our systems makes them more robust and resilient. Being aware of these issues can help you to identify and find potential sources of problems during development, as well as debug them in production.

Distributed Systems Abstractions

When talking about programming languages, we use common terminology and define our programs in terms of functions, operators, classes, variables, and pointers.

Having a common vocabulary helps us to avoid inventing new words every time we describe anything. The more precise and less ambiguous our definitions are, the easier it is for our listeners to understand us.

Before we move to algorithms, we first have to cover the distributed systems vocabulary: definitions you'll frequently encounter in talks, books, and papers.

Links

Networks are not reliable: messages can get lost, delayed, and reordered. Now, with this thought in our minds, we will try to build several communication protocols. We'll start with the least reliable and robust ones, identifying the states they can be in, and figuring out the possible additions to the protocol that can provide better guarantees.

Fair-loss link

We can start with two *processes*, connected with a *link*. Processes can send messages to each other, as shown in Figure 8-2. Any communication medium is imperfect, and messages can get lost or delayed.

Let's see what kind of guarantees we can get. After the message `M` is sent, from the senders' perspective, it can be in one of the following states:

- Not *yet* delivered to process `B` (but will be, at some point in time)
- Irrecoverably lost during transport
- Successfully delivered to the remote process

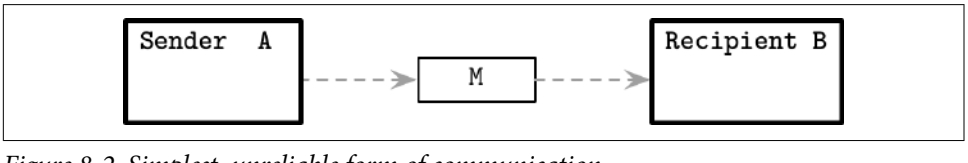


Figure 8-2. Simplest, unreliable form of communication

Notice that the sender does not have any way to find out if the message is already delivered. In distributed systems terminology, this kind of link is called *fair-loss*. The properties of this kind of link are:

Fair loss
: If both sender and recipient are correct and the sender keeps retransmitting the message infinitely many times, it will eventually be delivered.[3]

Finite duplication
: Sent messages won't be delivered infinitely many times.

No creation
: A link will not come up with messages; in other words, it won't deliver the message that was never sent.

A fair-loss link is a useful abstraction and a first building block for communication protocols with strong guarantees. We can assume that this link is not losing messages between communicating parties *systematically* and doesn't create new messages. But, at the same time, we cannot entirely rely on it. This might remind you of the User Datagram Protocol (UDP), which allows us to send messages from one process to the other, but does not have reliable delivery semantics on the protocol level.

Message acknowledgments

To improve the situation and get more clarity in terms of message status, we can introduce *acknowledgments*: a way for the recipient to notify the sender that it has received the message. For that, we need to use bidirectional communication channels and add some means that allow us to distinguish differences between the messages; for example, *sequence numbers*, which are unique monotonically increasing message identifiers.

It is enough to have a *unique* identifier for every message. Sequence numbers are just a particular case of a unique identifier, where we achieve uniqueness by drawing identifiers from a counter. When using hash algorithms to identify messages uniquely, we should account for possible collisions and make sure we can still disambiguate messages.

Now, process `A` can send a message `M(n)`, where `n` is a monotonically increasing message counter. As soon as `B` receives the message, it sends an acknowledgment `ACK(n)` back to A. Figure 8-3 shows this form of communication.

3 A more precise definition is that if a correct process A sends a message to a correct process B infinitely often, it will be delivered infinitely often (CACHIN11).

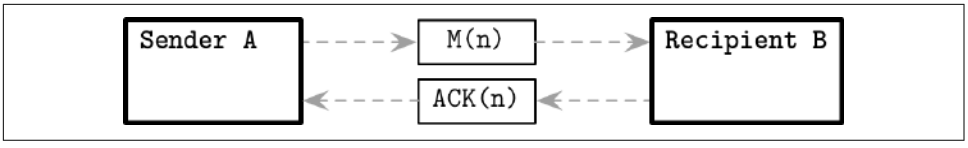


Figure 8-3. Sending a message with an acknowledgment

The acknowledgment, as well as the original message, may get lost on the way. The number of states the message can be in changes slightly. Until `A` receives an acknowledgment, the message is still in one of the three states we mentioned previously, but as soon as `A` receives the acknowledgment, it can be confident that the message is delivered to `B`.

Message retransmits

Adding acknowledgments is *still* not enough to call this communication protocol reliable: a sent message may still get lost, or the remote process may fail before acknowledging it. To solve this problem and provide delivery guarantees, we can try *retransmits* instead. Retransmits are a way for the sender to retry a potentially failed operation. We say *potentially* failed, because the sender doesn't really know whether it has failed or not, since the type of link we're about to discuss does *not* use acknowledgments.

After process `A` sends message `M`, it waits until timeout `T` is triggered and tries to send the same message again. Assuming the link between processes stays intact, network partitions between the processes are not infinite, and not *all* packets are lost, we can state that, from the sender's perspective, the message is either not *yet* delivered to process `B` or is successfully delivered to process `B`. Since `A` keeps trying to send the message, we can say that it *cannot* get irrecoverably lost during transport.

In distributed systems terminology, this abstraction is called a *stubborn link*. It's called stubborn because the sender keeps resending the message again and again indefinitely, but, since this sort of abstraction would be highly impractical, we need to combine retries with acknowledgments.

Problem with retransmits

Whenever we send the message, until we receive an acknowledgment from the remote process, we do not know whether it has already been processed, it will be processed shortly, it has been lost, or the remote process has crashed before receiving it—any one of these states is possible. We can retry the operation and send the message again, but this can result in message duplicates. Processing duplicates is only safe if the operation we're about to perform is idempotent.

An *idempotent* operation is one that can be executed multiple times, yielding the same result without producing additional side effects. For example, a server shut-

down operation can be idempotent, the first call initiates the shutdown, and all subsequent calls do not produce any additional effects.

If every operation was idempotent, we could think less about delivery semantics, rely more on retransmits for fault tolerance, and build systems in an entirely reactive way: triggering an action as a response to some signal, without causing unintended side effects. However, operations are not necessarily idempotent, and merely assuming that they are might lead to cluster-wide side effects. For example, charging a customer's credit card is not idempotent, and charging it multiple times is definitely undesirable.

Idempotence is particularly important in the presence of partial failures and network partitions, since we cannot always find out the exact status of a remote operation—whether it has succeeded, failed, or will be executed shortly—and we just have to wait longer. Since guaranteeing that each executed operation is idempotent is an unrealistic requirement, we need to provide guarantees *equivalent* to idempotence without changing the underlying operation semantics. To achieve this, we can use *deduplication* and avoid processing messages more than once.

Message order

Unreliable networks present us with two problems: messages can arrive out of order and, because of retransmits, some messages may arrive more than once. We have already introduced sequence numbers, and we can use these message identifiers on the recipient side to ensure *first-in, first-out* (FIFO) ordering. Since every message has a sequence number, the receiver can track:

- $n_{consecutive}$, specifying the highest sequence number, up to which it has seen all messages. Messages up to this number can be put back in order.
- $n_{processed}$, specifying the highest sequence number, up to which messages were put back in their original order and *processed*. This number can be used for deduplication.

If the received message has a nonconsecutive sequence number, the receiver puts it into the reordering buffer. For example, it receives a message with a sequence number `5` after receiving one with `3`, and we know that `4` is still missing, so we need to put `5` aside until `4` comes, and we can reconstruct the message order. Since we're building on top of a fair-loss link, we assume that messages between $n_{consecutive}$ and n_{max_seen} will eventually be delivered.

The recipient can safely discard the messages with sequence numbers up to $n_{consecutive}$ that it receives, since they're guaranteed to be already delivered.

Deduplication works by checking if the message with a sequence number `n` has already been *processed* (passed down the stack by the receiver) and discarding already processed messages.

In distributed systems terms, this type of link is called a *perfect link*, which provides the following guarantees [CACHIN11]:

Reliable delivery
: Every message sent *once* by the correct process `A` to the correct process `B`, will *eventually* be delivered.

No duplication
: No message is delivered more than once.

No creation
: Same as with other types of links, it can only deliver the messages that were actually sent.

This might remind you of the TCP[4] protocol (however, reliable delivery in TCP is guaranteed only in the scope of a single session). Of course, this model is just a simplified representation we use for illustration purposes only. TCP has a much more sophisticated model for dealing with acknowledgments, which groups acknowledgments and reduces the protocol-level overhead. In addition, TCP has selective acknowledgments, flow control, congestion control, error detection, and many other features that are out of the scope of our discussion.

Exactly-once delivery

> There are only two hard problems in distributed systems: 2. Exactly-once delivery 1. Guaranteed order of messages 2. Exactly-once delivery.
>
> —Mathias Verraes

There have been many discussions about whether or not *exactly-once delivery* is possible. Here, semantics and precise wording are essential. Since there might be a link failure preventing the message from being delivered from the first try, most of the real-world systems employ *at-least-once delivery*, which ensures that the sender retries until it receives an acknowledgment, otherwise the message is not considered to be received. Another delivery semantic is *at-most-once*: the sender sends the message and doesn't expect any delivery confirmation.

The TCP protocol works by breaking down messages into packets, transmitting them one by one, and stitching them back together on the receiving side. TCP might attempt to retransmit some of the packets, and more than one transmission attempt

4 See *https://databass.dev/links/53*.

may succeed. Since TCP marks each packet with a sequence number, even though some packets were transmitted more than once, it can deduplicate the packets and guarantee that the recipient will see the message and *process* it only once. In TCP, this guarantee is valid only for a *single session*: if the message is acknowledged and processed, but the sender didn't receive the acknowledgment before the connection was interrupted, the application is not aware of this delivery and, depending on its logic, it might attempt to send the message once again.

This means that exactly-once *processing* is what's interesting here since duplicate *deliveries* (or packet transmissions) have no side effects and are merely an artifact of the best effort by the link. For example, if the database node has only *received* the record, but hasn't *persisted* it, delivery has occurred, but it'll be of no use unless the record can be retrieved (in other words, unless it was both delivered and processed).

For the exactly-once guarantee to hold, nodes should have a *common knowledge* [HALPERN90]: everyone knows about some fact, and everyone knows that everyone else also knows about that fact. In simplified terms, nodes have to agree on the state of the record: both nodes agree that it either *was* or *was not* persisted. As you will see later in this chapter, this is theoretically impossible, but in practice we still use this notion by relaxing coordination requirements.

Any misunderstanding about whether or not exactly-once delivery is possible most likely comes from approaching the problem from different protocol and abstraction levels and the definition of "delivery." It's not possible to build a reliable link without ever transferring any message more than once, but we can create the illusion of exactly-once delivery from the sender's perspective by *processing* the message once and ignoring duplicates.

Now, as we have established the means for reliable communication, we can move ahead and look for ways to achieve uniformity and agreement between processes in the distributed system.

Two Generals' Problem

One of the most prominent descriptions of an agreement in a distributed system is a thought experiment widely known as the *Two Generals' Problem*.

This thought experiment shows that it is impossible to achieve an agreement between two parties if communication is *asynchronous* in the presence of link failures. Even though TCP exhibits properties of a perfect link, it's important to remember that perfect links, despite the name, do not guarantee *perfect* delivery. They also can't guarantee that participants will be alive the whole time, and are concerned only with transport.

Imagine two armies, led by two generals, preparing to attack a fortified city. The armies are located on two sides of the city and can succeed in their siege only if they attack simultaneously.

The generals can communicate by sending messengers, and already have devised an attack plan. The only thing they now have to agree on is whether or not to carry out the plan. Variants of this problem are when one of the generals has a higher rank, but needs to make sure the attack is coordinated; or that the generals need to agree on the exact time. These details do not change the problem definition: the generals have to come to an agreement.

The army generals only have to agree on the fact that they both will proceed with the attack. Otherwise, the attack cannot succeed. General `A` sends a message `MSG(N)`, stating an intention to proceed with the attack at a specified time, *if* the other party agrees to proceed as well.

After `A` sends the messenger, he doesn't know whether the messenger has arrived or not: the messenger can get captured and fail to deliver the message. When general `B` receives the message, he has to send an acknowledgment `ACK(MSG(N))`. Figure 8-4 shows that a message is sent one way and acknowledged by the other party.

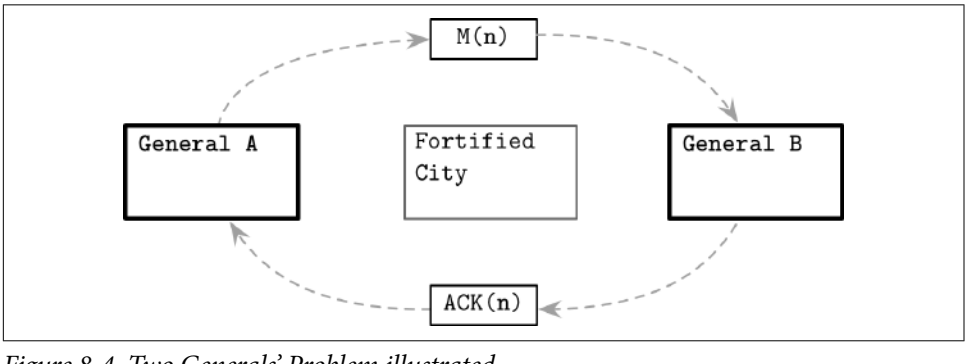


Figure 8-4. Two Generals' Problem illustrated

The messenger carrying this acknowledgment might get captured or fail to deliver it, as well. `B` doesn't have any way of knowing if the messenger has successfully delivered the acknowledgment.

To be sure about it, `B` has to wait for `ACK(ACK(MSG(N)))`, a second-order acknowledgment stating that `A` received an acknowledgment for the acknowledgment.

No matter how many further confirmations the generals send to each other, they will always be one `ACK` away from knowing if they can safely proceed with the attack. The generals are doomed to wonder if the message carrying this last acknowledgment has reached the destination.

Notice that we did not make any timing assumptions: communication between generals is fully asynchronous. There is no upper time bound set on how long the generals can take to respond.

FLP Impossibility

In a paper by Fisher, Lynch, and Paterson, the authors describe a problem famously known as the *FLP Impossibility Problem* [FISCHER85] (derived from the first letters of authors' last names), wherein they discuss a form of consensus in which processes start with an initial value and attempt to agree on a new value. After the algorithm completes, this new value has to be the same for all nonfaulty processes.

Reaching an agreement on a specific value is straightforward if the network is entirely reliable; but in reality, systems are prone to many different sorts of failures, such as message loss, duplication, network partitions, and slow or crashed processes.

A consensus protocol describes a system that, given multiple processes starting at its *initial state*, brings all of the processes to the *decision state*. For a consensus protocol to be correct, it has to preserve three properties:

Agreement
: The decision the protocol arrives at has to be unanimous: each process decides on some value, and this has to be the same for all processes. Otherwise, we have not reached a consensus.

Validity
: The agreed value has to be *proposed* by one of the participants, which means that the system should not just "come up" with the value. This also implies nontriviality of the value: processes should not always decide on some predefined default value.

Termination
: An agreement is final only if there are no processes that did not reach the decision state.

[FISCHER85] assumes that processing is entirely asynchronous; there's no shared notion of time between the processes. Algorithms in such systems cannot be based on timeouts, and there's no way for a process to find out whether the other process has crashed or is simply running too slow. The paper shows that, given these assumptions, there exists no protocol that can guarantee consensus in a bounded time. No completely asynchronous consensus algorithm can tolerate the unannounced crash of even a single remote process.

If we do not consider an upper time bound for the process to complete the algorithm steps, process failures can't be reliably detected, and there's no deterministic algorithm to reach a consensus.

However, FLP Impossibility does not mean we have to pack our things and go home, as reaching consensus is not possible. It only means that we cannot always reach consensus in an asynchronous system in bounded time. In practice, systems exhibit at least some degree of synchrony, and the solution to this problem requires a more refined model.

System Synchrony

From FLP Impossibility, you can see that the timing assumption is one of the critical characteristics of the distributed system. In an *asynchronous system*, we do not know the relative speeds of processes, and cannot guarantee message delivery in a bounded time or a particular order. The process might take indefinitely long to respond, and process failures can't always be reliably detected.

The main criticism of asynchronous systems is that these assumptions are not realistic: processes can't have *arbitrarily* different processing speeds, and links don't take *indefinitely* long to deliver messages. Relying on time both simplifies reasoning and helps to provide upper-bound timing guarantees.

It is not always possible to solve a consensus problem in an asynchronous model [FISCHER85]. Moreover, designing an efficient asynchronous algorithm is not always achievable, and for some tasks the practical solutions are more likely to be time-dependent [ARJOMANDI83].

These assumptions can be loosened up, and the system can be considered to be *synchronous*. For that, we introduce the notion of timing. It is much easier to reason about the system under the synchronous model. It assumes that processes are progressing at comparable rates, that transmission delays are bounded, and message delivery cannot take arbitrarily long.

A synchronous system can also be represented in terms of synchronized process-local clocks: there is an upper time bound in time difference between the two process-local time sources [CACHIN11].

Designing systems under a synchronous model allows us to use timeouts. We can build more complex abstractions, such as leader election, consensus, failure detection, and many others on top of them. This makes the best-case scenarios more robust, but results in a failure if the timing assumptions don't hold up. For example, in the Raft consensus algorithm (see Chapter 14), we may end up with multiple processes believing they're leaders, which is resolved by forcing the lagging process to accept the other process as a leader; failure-detection algorithms (see Chapter 9) can wrongly identify a live process as failed or vice versa. When designing our systems, we should make sure to consider these possibilities.

Properties of both asynchronous and synchronous models can be combined, and we can think of a system as *partially synchronous*. A partially synchronous system exhibits some of the properties of the synchronous system, but the bounds of message delivery, clock drift, and relative processing speeds might not be exact and hold only *most of the time* [DWORK88].

Synchrony is an essential property of the distributed system: it has an impact on performance, scalability, and general solvability, and has many factors necessary for the correct functioning of our systems. Some of the algorithms we discuss in this book operate under the assumptions of synchronous systems.

Failure Models

We keep mentioning *failures*, but so far it has been a rather broad and generic concept that might capture many meanings. Similar to how we can make different timing assumptions, we can assume the presence of different types of failures. A *failure model* describes exactly how processes can crash in a distributed system, and algorithms are developed using these assumptions. For example, we can assume that a process can crash and never recover, or that it is expected to recover after some time passes, or that it can fail by spinning out of control and supplying incorrect values.

In distributed systems, processes rely on one another for executing an algorithm, so failures can result in incorrect execution across the whole system.

We'll discuss multiple failure models present in distributed systems, such as *crash*, *omission*, and *arbitrary* faults. This list is not exhaustive, but it covers most of the cases applicable and important in real-life systems.

Crash Faults

Normally, we expect the process to be executing all steps of an algorithm correctly. The simplest way for a process to crash is by *stopping* the execution of any further steps required by the algorithm and not sending any messages to other processes. In other words, the process *crashes*. Most of the time, we assume a *crash-stop* process abstraction, which prescribes that, once the process has crashed, it remains in this state.

This model does not assume that it is impossible for the process to recover, and does not discourage recovery or try to prevent it. It only means that the algorithm *does not rely* on recovery for correctness or liveness. Nothing prevents processes from recovering, catching up with the system state, and participating in the *next* instance of the algorithm.

Failed processes are not able to continue participating in the current round of negotiations during which they failed. Assigning the recovering process a new, different

identity does not make the model equivalent to crash-recovery (discussed next), since most algorithms use predefined lists of processes and clearly define failure semantics in terms of how many failures they can tolerate [CACHIN11].

Crash-recovery is a different process abstraction, under which the process stops executing the steps required by the algorithm, but recovers at a later point and tries to execute further steps. The possibility of recovery requires introducing a durable state and recovery protocol into the system [SKEEN83]. Algorithms that allow crash-recovery need to take all possible recovery states into consideration, since the recovering process may attempt to continue execution from the last step known to it.

Algorithms, aiming to exploit recovery, have to take both state and identity into account. Crash-recovery, in this case, can also be viewed as a special case of omission failure, since from the other process's perspective there's no distinction between the process that was unreachable and the one that has crashed and recovered.

Omission Faults

Another failure model is *omission fault*. This model assumes that the process skips some of the algorithm steps, or is not able to execute them, or this execution is not visible to other participants, or it cannot send or receive messages to and from other participants. Omission fault captures network partitions between the processes caused by faulty network links, switch failures, or network congestion. Network partitions can be represented as omissions of messages between individual processes or process groups. A crash can be simulated by completely omitting any messages to and from the process.

When the process is operating slower than the other participants and sends responses much later than expected, for the rest of the system it may look like it is forgetful. Instead of stopping completely, a slow node attempts to send its results out of sync with other nodes.

Omission failures occur when the algorithm that was supposed to execute certain steps either skips them or the results of this execution are not visible. For example, this may happen if the message is lost on the way to the recipient, and the sender fails to send it again and continues to operate as if it was successfully delivered, even though it was irrecoverably lost. Omission failures can also be caused by intermittent hangs, overloaded networks, full queues, etc.

Arbitrary Faults

The hardest class of failures to overcome is *arbitrary* or *Byzantine* faults: a process continues executing the algorithm steps, but in a way that contradicts the algorithm (for example, if a process in a consensus algorithm decides on a value that no other participant has ever proposed).

Such failures can happen due to bugs in software, or due to processes running different versions of the algorithm, in which case failures are easier to find and understand. It can get much more difficult when we do not have control over all processes, and one of the processes is intentionally misleading other processes.

You might have heard of Byzantine fault tolerance from the airspace industry: airplane and spacecraft systems do not take responses from subcomponents at face value and cross-validate their results. Another widespread application is cryptocurrencies [GILAD17], where there is no central authority, different parties control the nodes, and adversary participants have a material incentive to forge values and attempt to game the system by providing faulty responses.

Handling Failures

We can *mask* failures by forming process groups and introducing redundancy into the algorithm: even if one of the processes fails, the user will not notice this failure [CHRISTIAN91].

There might be some performance penalty related to failures: normal execution relies on processes being responsive, and the system has to fall back to the slower execution path for error handling and correction. Many failures can be prevented on the software level by code reviews, extensive testing, ensuring message delivery by introducing timeouts and retries, and making sure that steps are executed in order locally.

Most of the algorithms we're going to cover here assume the crash-failure model and work around failures by introducing redundancy. These assumptions help to create algorithms that perform better and are easier to understand and implement.

Summary

In this chapter, we discussed some of the distributed systems terminology and introduced some basic concepts. We've talked about the inherent difficulties and complications caused by the unreliability of the system components: links may fail to deliver messages, processes may crash, or the network may get partitioned.

This terminology should be enough for us to continue the discussion. The rest of the book talks about the *solutions* commonly used in distributed systems: we think back to what can go wrong and see what options we have available.

Further Reading

If you'd like to learn more about the concepts mentioned in this chapter, you can refer to the following sources:

Distributed systems abstractions, failure models, and timing assumptions

Lynch, Nancy A. 1996. *Distributed Algorithms*. San Francisco: Morgan Kaufmann.

Tanenbaum, Andrew S. and Maarten van Steen. 2006. *Distributed Systems: Principles and Paradigms* (2nd Ed). Boston: Pearson.

Cachin, Christian, Rachid Guerraoui, and Lus Rodrigues. 2011. *Introduction to Reliable and Secure Distributed Programming* (2nd Ed.). New York: Springer.

CHAPTER 13
Distributed Transactions

To maintain order in a distributed system, we have to guarantee at least some consistency. In Chapter 11, we talked about single-object, single-operation consistency models that help us to reason about the individual operations. However, in databases we often need to execute *multiple* operations atomically.

Atomic operations are explained in terms of state transitions: the database was in state `A` before a particular transaction was started; by the time it finished, the state went from `A` to `B`. In operation terms, this is simple to understand, since transactions have no predetermined attached state. Instead, they apply operations to data records starting at *some* point in time. This gives us some flexibility in terms of scheduling and execution: transactions can be reordered and even retried.

The main focus of transaction processing is to determine permissible *histories*, to model and represent possible interleaving execution scenarios. History, in this case, represents a dependency graph: which transactions have been executed prior to execution of the current transaction. History is said to be *serializable* if it is equivalent (i.e., has the same dependency graph) to *some* history that executes these transactions sequentially. You can review concepts of histories, their equivalence, serializability, and other concepts in Chapter 5. Generally, this chapter is a distributed systems counterpart of Chapter 5, where we discussed node-local transaction processing.

Single-partition transactions involve the pessimistic (lock-based or tracking) or optimistic (try and validate) concurrency control schemes that we discussed in Chapter 5, but neither one of these approaches solves the problem of multipartition transactions, which require coordination between different servers, distributed commit, and rollback protocols.

Generally speaking, when transferring money from one account to another, you'd like to both credit the first account and debit the second one *simultaneously*. However, if

we break down the transaction into individual steps, even debiting or crediting doesn't look atomic at first sight: we need to read the old balance, add or subtract the required amount, and save this result. Each one of these substeps involves several operations: the node receives a request, parses it, locates the data on disk, makes a write and, finally, acknowledges it. Even this is a rather high-level view: to execute a simple write, we have to perform hundreds of small steps.

This means that we have to first *execute* the transaction and only then make its results *visible*. But let's first define what transactions are. A *transaction* is a set of operations, an atomic unit of execution. Transaction atomicity implies that all its results become visible or none of them do. For example, if we modify several rows, or even tables in a single transaction, either all or none of the modifications will be applied.

To ensure atomicity, transactions should be *recoverable*. In other words, if the transaction cannot complete, is aborted, or times out, its results have to be rolled back completely. A nonrecoverable, partially executed transaction can leave the database in an inconsistent state. In summary, in case of unsuccessful transaction execution, the database state has to be reverted to its previous state, as if this transaction was never tried in the first place.

Another important aspect is network partitions and node failures: nodes in the system fail and recover independently, but their states have to remain consistent. This means that the atomicity requirement holds not only for the local operations, but also for operations executed on other nodes: changes have to be durably propagated to all of the nodes involved in the transaction or none of them [LAMPSON79].

Making Operations Appear Atomic

To make multiple operations appear atomic, especially if some of them are remote, we need to use a class of algorithms called *atomic commitment*. Atomic commitment doesn't allow disagreements between the participants: a transaction *will not* commit if even one of the participants votes against it. At the same time, this means that *failed* processes have to reach the same conclusion as the rest of the cohort. Another important implication of this fact is that atomic commitment algorithms do not work in the presence of Byzantine failures: when the process lies about its state or decides on an arbitrary value, since it contradicts unanimity [HADZILACOS0].

The problem that atomic commitment is trying to solve is reaching an agreement on whether or not to execute the proposed transaction. Cohorts cannot choose, influence, or change the proposed transaction or propose any alternative: they can only give their vote on whether or not they are willing to execute it [ROBINSON08].

Atomic commitment algorithms do not set strict requirements for the semantics of transaction *prepare*, *commit*, or *rollback* operations. Database implementers have to decide on:

- When the data is considered ready to commit, and they're just a pointer swap away from making the changes public.
- How to perform the commit itself to make transaction results visible in the shortest timeframe possible.
- How to roll back the changes made by the transaction if the algorithm decides not to commit.

We discussed node-local implementations of these processes in Chapter 5.

Many distributed systems use atomic commitment algorithms—for example, MySQL (for distributed transactions) and Kafka (for producer and consumer interaction MEHTA17).

In databases, distributed transactions are executed by the component commonly known as a *transaction manager*. The transaction manager is a subsystem responsible for scheduling, coordinating, executing, and tracking transactions. In a distributed environment, the transaction manager is responsible for ensuring that node-local visibility guarantees are consistent with the visibility prescribed by distributed atomic operations. In other words, transactions commit in all partitions, and for all replicas.

We will discuss two atomic commitment algorithms: two-phase commit, which solves a commitment problem, but doesn't allow for failures of the coordinator process; and three-phase commit [SKEEN83], which solves a *nonblocking atomic commitment* problem,[1] and allows participants proceed even in case of coordinator failures [BABAOGLU93].

Two-Phase Commit

Let's start with the most straightforward protocol for a distributed commit that allows multipartition *atomic* updates. (For more information on partitioning, you can refer to "Database Partitioning" on page 66.) *Two-phase commit* (2PC) is usually discussed in the context of database transactions. 2PC executes in two phases. During the first phase, the decided value is distributed, and votes are collected. During the second phase, nodes just flip the switch, making the results of the first phase visible.

2PC assumes the presence of a *leader* (or *coordinator*) that holds the state, collects votes, and is a primary point of reference for the agreement round. The rest of the nodes are called *cohorts*. Cohorts, in this case, are usually partitions that operate over disjoint datasets, against which transactions are performed. The coordinator and

1 The fine print says "assuming a highly reliable network." In other words, a network that precludes partitions [ALHOUMAILY10]. Implications of this assumption are discussed in the paper's section about algorithm description.

every cohort keep local operation logs for each executed step. Participants vote to accept or reject some *value*, proposed by the coordinator. Most often, this value is an identifier of the distributed transaction that has to be executed, but 2PC can be used in other contexts as well.

The coordinator can be a node that received a request to execute the transaction, or it can be picked at random, using a leader-election algorithm, assigned manually, or even fixed throughout the lifetime of the system. The protocol does not place restrictions on the coordinator role, and the role can be transferred to another participant for reliability or performance.

As the name suggests, a two-phase commit is executed in two steps:

Prepare
: The coordinator notifies cohorts about the new transaction by sending a `Propose` message. Cohorts make a decision on whether or not they can commit the part of the transaction that applies to them. If a cohort decides that it can commit, it notifies the coordinator about the positive vote. Otherwise, it responds to the coordinator, asking it to abort the transaction. All decisions taken by cohorts are persisted in the coordinator log, and each cohort keeps a copy of its decision locally.

Commit/abort
: Operations within a transaction can change state across different partitions (each represented by a cohort). If even one of the cohorts votes to abort the transaction, the coordinator sends the `Abort` message to all of them. Only if all cohorts have voted positively does the coordinator send them a final `Commit` message.

This process is shown in Figure 13-1.

During the *prepare* phase, the coordinator distributes the proposed value and collects votes from the participants on whether or not this proposed value should be committed. Cohorts may choose to reject the coordinator's proposal if, for example, another conflicting transaction has already committed a different value.

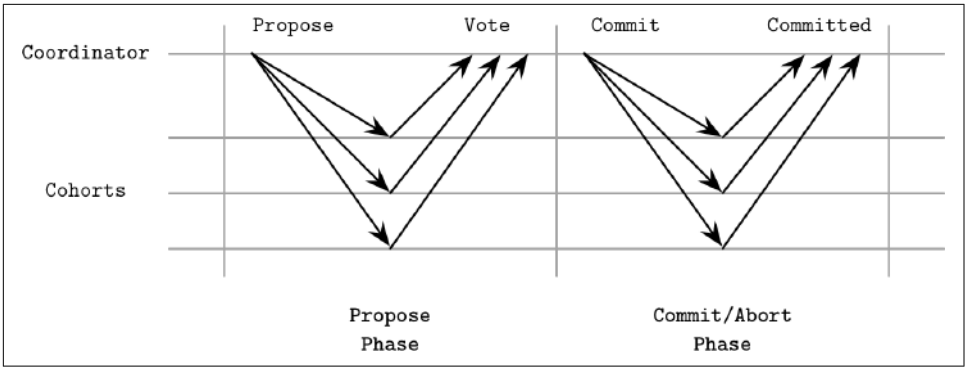


Figure 13-1. Two-phase commit protocol. During the first phase, cohorts are notified about the new transaction. During the second phase, the transaction is committed or aborted.

After the coordinator has collected the votes, it can make a decision on whether to *commit* the transaction or *abort* it. If all cohorts have voted positively, it decides to commit and notifies them by sending a `Commit` message. Otherwise, the coordinator sends an `Abort` message to all cohorts and the transaction gets rolled back. In other words, if one node rejects the proposal, the whole round is aborted.

During each step the coordinator and cohorts have to write the results of each operation to durable storage to be able to reconstruct the state and recover in case of local failures, and be able to forward and replay results for other participants.

In the context of database systems, each 2PC round is usually responsible for a single transaction. During the *prepare* phase, transaction contents (operations, identifiers, and other metadata) are transferred from the coordinator to the cohorts. The transaction is executed by the cohorts locally and is left in a *partially committed* state (sometimes called *precommitted*), making it ready for the coordinator to finalize execution during the next phase by either committing or aborting it. By the time the transaction commits, its contents are already stored durably on all other nodes [BERNSTEIN09].

Cohort Failures in 2PC

Let's consider several failure scenarios. For example, as Figure 13-2 shows, if one of the cohorts fails during the *propose* phase, the coordinator cannot proceed with a commit, since it requires all votes to be positive. If one of the cohorts is unavailable, the coordinator will abort the transaction. This requirement has a negative impact on availability: failure of a single node can prevent transactions from happening. Some systems, for example, Spanner (see "Distributed Transactions with Spanner" on page 64), perform 2PC over Paxos groups rather than individual nodes to improve protocol availability.

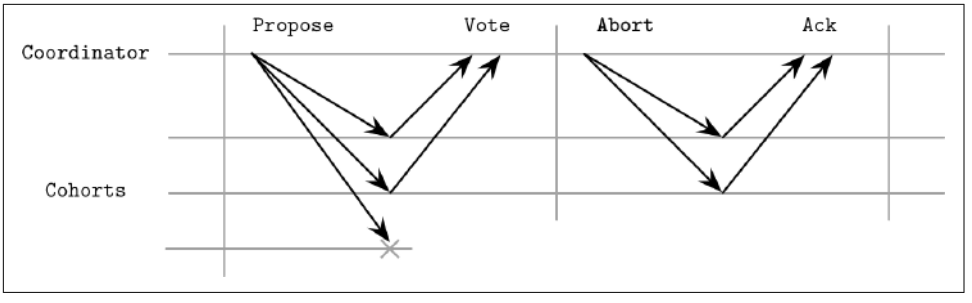


Figure 13-2. Cohort failure during the propose phase

The main idea behind 2PC is a *promise* by a cohort that, once it has positively responded to the proposal, it will not go back on its decision, so only the coordinator can abort the transaction.

If one of the cohorts has failed *after* accepting the proposal, it has to learn about the actual outcome of the vote before it can serve values correctly, since the coordinator might have aborted the commit due to the other cohorts' decisions. When a cohort node recovers, it has to get up to speed with a final coordinator decision. Usually, this is done by persisting the decision log on the coordinator side and replicating decision values to the failed participants. Until then, the cohort cannot serve requests because it is in an inconsistent state.

Since the protocol has multiple spots where processes are waiting for the other participants (when the coordinator collects votes, or when the cohort is waiting for the commit/abort phase), link failures might lead to message loss, and this wait will continue indefinitely. If the coordinator does not receive a response from the replica during the propose phase, it can trigger a timeout and abort the transaction.

Coordinator Failures in 2PC

If one of the cohorts does not receive a commit or abort command from the coordinator during the second phase, as shown in Figure 13-3, it should attempt to find out which decision was made by the coordinator. The coordinator might have decided upon the value but wasn't able to communicate it to the particular replica. In such cases, information about the decision can be replicated from the peers' transaction logs or from the backup coordinator. Replicating commit decisions is safe since it's always unanimous: the whole point of 2PC is to either commit or abort on all sites, and commit on one cohort implies that all other cohorts have to commit.

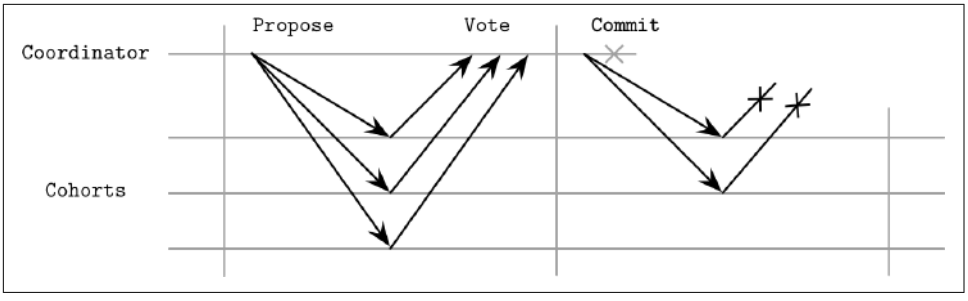


Figure 13-3. Coordinator failure after the propose phase

During the first phase, the coordinator collects votes and, subsequently, promises from cohorts, that they will wait for its explicit commit or abort command. If the coordinator fails after collecting the votes, but before broadcasting vote results, the cohorts end up in a state of uncertainty. This is shown in Figure 13-4. Cohorts do not know what precisely the coordinator has decided, and whether or not any of the participants (potentially also unreachable) might have been notified about the transaction results [BERNSTEIN87].

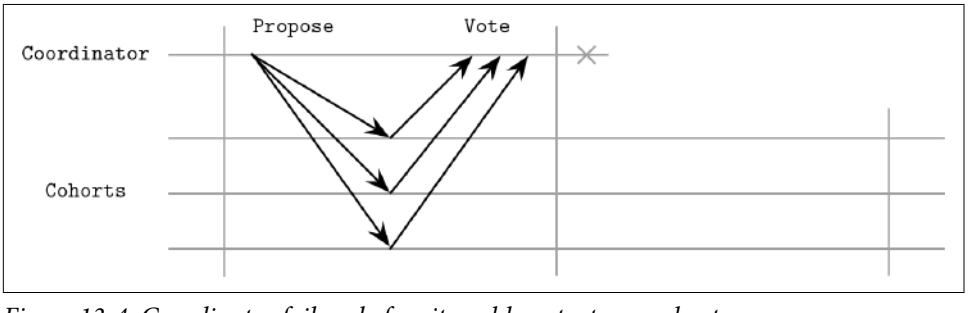


Figure 13-4. Coordinator failure before it could contact any cohorts

Inability of the coordinator to proceed with a commit or abort leaves the cluster in an undecided state. This means that cohorts will not be able to learn about the final decision in case of a permanent coordinator failure. Because of this property, we say that 2PC is a *blocking* atomic commitment algorithm. If the coordinator never recovers, its replacement has to collect votes for a given transaction again, and proceed with a final decision.

Many databases use 2PC: MySQL, PostgreSQL, MongoDB,[2] and others. Two-phase commit is often used to implement distributed transactions because of its simplicity (it is easy to reason about, implement, and debug) and low overhead (message com-

2 However, the documentation says that as of v3.6, 2PC provides only transaction-*like* semantics: *https://databass.dev/links/7*.

plexity and the number of round-trips of the protocol are low). It is important to implement proper recovery mechanisms and have backup coordinator nodes to reduce the chance of the failures just described.

Three-Phase Commit

To make an atomic commitment protocol robust against coordinator failures and avoid undecided states, the three-phase commit (3PC) protocol adds an extra step, and timeouts on *both* sides that can allow cohorts to proceed with either commit or abort in the event of coordinator failure, depending on the system state. 3PC assumes a synchronous model and that communication failures are not possible [BABAOGLU93].

3PC adds a *prepare* phase before the commit/abort step, which communicates cohort states collected by the coordinator during the propose phase, allowing the protocol to carry on even if the coordinator fails. All other properties of 3PC and a requirement to have a coordinator for the round are similar to its two-phase sibling. Another useful addition to 3PC is timeouts on the cohort side. Depending on which step the process is currently executing, either a commit or abort decision is forced on timeout.

As Figure 13-5 shows, the three-phase commit round consists of three steps:

Propose
: The coordinator sends out a proposed value and collects the votes.

Prepare
: The coordinator notifies cohorts about the vote results. If the vote has passed and all cohorts have decided to commit, the coordinator sends a `Prepare` message, instructing them to prepare to commit. Otherwise, an `Abort` message is sent and the round completes.

Commit
: Cohorts are notified by the coordinator to commit the transaction.

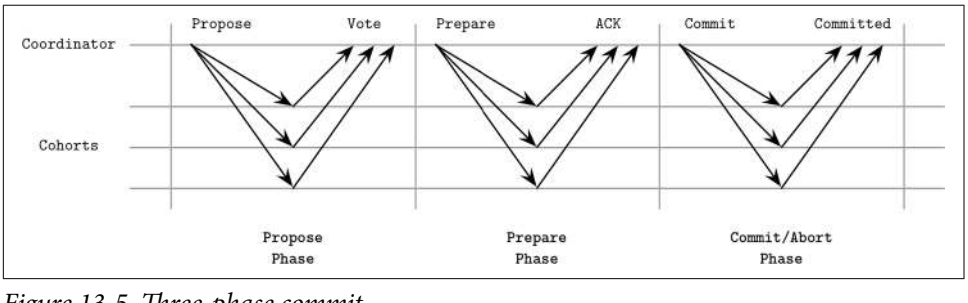


Figure 13-5. Three-phase commit

During the *propose* step, similar to 2PC, the coordinator distributes the proposed value and collects votes from cohorts, as shown in Figure 13-5. If the coordinator crashes during this phase and the operation times out, or if one of the cohorts votes negatively, the transaction will be aborted.

After collecting the votes, the coordinator makes a decision. If the coordinator decides to proceed with a transaction, it issues a `Prepare` command. It may happen that the coordinator cannot distribute prepare messages to all cohorts or it fails to receive their acknowledgments. In this case, cohorts may abort the transaction after timeout, since the algorithm hasn't moved all the way to the *prepared* state.

As soon as all the cohorts successfully move into the prepared state and the coordinator has received their prepare acknowledgments, the transaction will be committed if either side fails. This can be done since all participants at this stage have the same view of the state.

During *commit*, the coordinator communicates the results of the *prepare* phase to all the participants, resetting their timeout counters and effectively finishing the transaction.

Coordinator Failures in 3PC

All state transitions are coordinated, and cohorts can't move on to the next phase until everyone is done with the previous one: the coordinator has to wait for the replicas to continue. Cohorts can eventually abort the transaction if they do not hear from the coordinator before the timeout, if they didn't move past the prepare phase.

As we discussed previously, 2PC cannot recover from coordinator failures, and cohorts may get stuck in a nondeterministic state until the coordinator comes back. 3PC avoids blocking the processes in this case and allows cohorts to proceed with a deterministic decision.

The worst-case scenario for the 3PC is a network partition, shown in Figure 13-6. Some nodes successfully move to the prepared state, and now can proceed with commit after the timeout. Some can't communicate with the coordinator, and will abort after the timeout. This results in a split brain: some nodes proceed with a commit and some abort, all according to the protocol, leaving participants in an inconsistent and contradictory state.

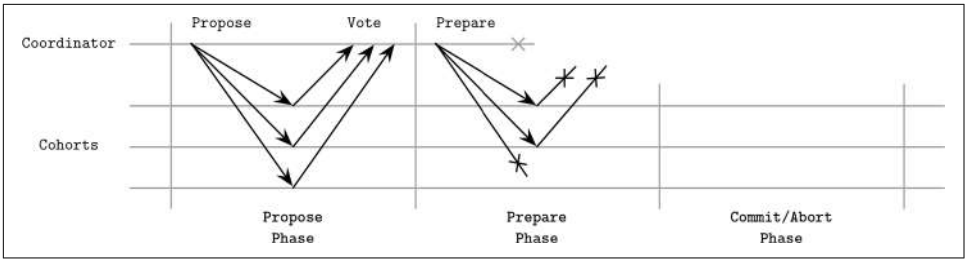


Figure 13-6. Coordinator failure during the second phase

While in theory 3PC does, to a degree, solve the problem with 2PC blocking, it has a larger message overhead, introduces potential contradictions, and does not work well in the presence of network partitions. This might be the primary reason 3PC is not widely used in practice.

Distributed Transactions with Calvin

We've already touched on the subject of synchronization costs and several ways around it. But there are other ways to reduce contention and the total amount of time during which transactions hold locks. One of the ways to do this is to let replicas agree on the execution order and transaction boundaries before acquiring locks and proceeding with execution. If we can achieve this, node failures do not cause transaction aborts, since nodes can recover state from other participants that execute the same transaction in parallel.

Traditional database systems execute transactions using two-phase locking or optimistic concurrency control and have no deterministic transaction order. This means that nodes have to be coordinated to preserve order. Deterministic transaction order removes coordination overhead during the execution phase and, since all replicas get the same inputs, they also produce equivalent outputs. This approach is commonly known as Calvin, a fast distributed transaction protocol [THOMSON12]. One of the prominent examples implementing distributed transactions using Calvin is FaunaDB.

To achieve deterministic order, Calvin uses a *sequencer*: an entry point for all transactions. The sequencer determines the order in which transactions are executed, and establishes a global transaction input sequence. To minimize contention and batch decisions, the timeline is split into *epochs*. The sequencer collects transactions and groups them into short time windows (the original paper mentions 10-millisecond batches), which also become replication units, so transactions do not have to be communicated separately.

As soon as a transaction batch is successfully replicated, sequencer forwards it to the *scheduler*, which orchestrates transaction execution. The scheduler uses a deterministic scheduling protocol that executes parts of transaction in parallel, while preserving the serial execution order specified by the sequencer. Since applying transaction to a specific state is guaranteed to produce only changes specified by the transaction and transaction order is predetermined, replicas do not have to further communicate with the sequencer.

Each transaction in Calvin has a *read set* (its dependencies, which is a collection of data records from the current database state required to execute it) and a *write set* (results of the transaction execution; in other words, its side effects). Calvin does not natively support transactions that rely on additional reads that would determine read and write sets.

A worker thread, managed by the scheduler, proceeds with execution in four steps:

1. It analyzes the transaction's read and write sets, determines node-local data records from the read set, and creates the list of *active* participants (i.e., ones that hold the elements of the write set, and will perform modifications on the data).
2. It collects the *local* data required to execute the transaction, in other words, the read set records that happen to reside on that node. The collected data records are forwarded to the corresponding *active* participants.
3. If this worker thread is executing on an active participant node, it receives data records forwarded from the other participants, as a counterpart of the operations executed during step 2.
4. Finally, it executes a batch of transactions, persisting results into local storage. It does not have to forward execution results to the other nodes, as they receive the same inputs for transactions and execute and persist results locally themselves.

A typical Calvin implementation colocates sequencer, scheduler, worker, and storage subsystems, as Figure 13-7 shows. To make sure that sequencers reach consensus on exactly which transactions make it into the current epoch/batch, Calvin uses the Paxos consensus algorithm (see Chapter 14) or asynchronous replication, in which a dedicated replica serves as a leader. While using a leader can improve latency, it comes with a higher cost of recovery as nodes have to reproduce the state of the failed leader in order to proceed.

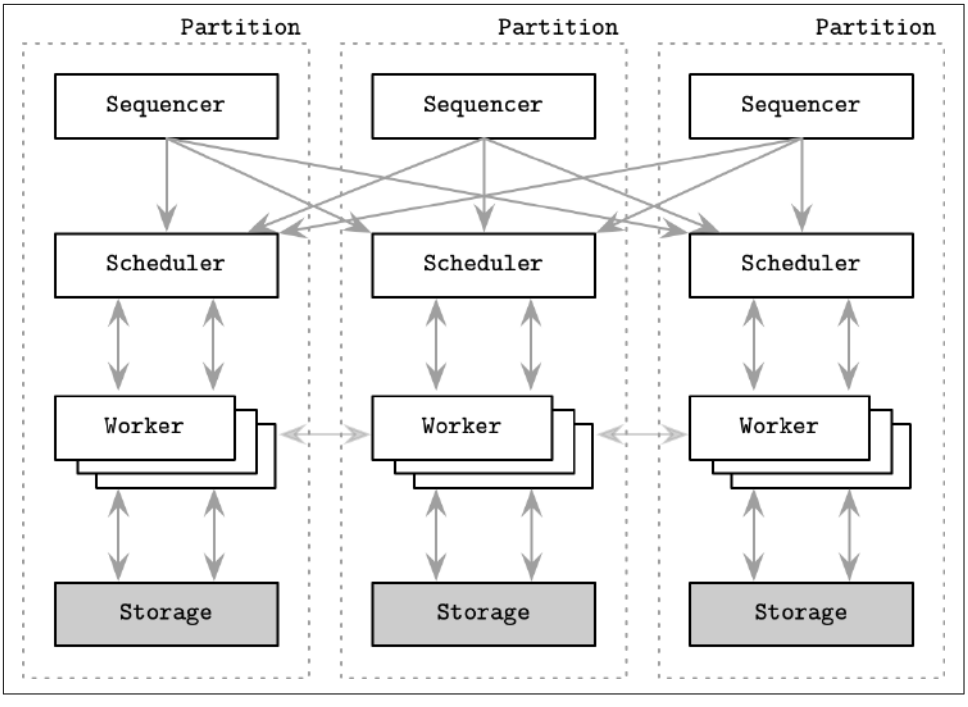


Figure 13-7. Calvin architecture

Distributed Transactions with Spanner

Calvin is often contrasted with another approach for distributed transaction management called Spanner [CORBETT12]. Its implementations (or derivatives) include several open source databases, most prominently CockroachDB and YugaByte DB. While Calvin establishes the global transaction execution order by reaching consensus on sequencers, Spanner uses two-phase commit over consensus groups per partition (in other words, per shard). Spanner has a rather complex setup, and we only cover high-level details in the scope of this book.

To achieve consistency and impose transaction order, Spanner uses *TrueTime*: a high-precision wall-clock API that also exposes an uncertainty bound, allowing local operations to introduce artificial slowdowns to wait for the uncertainty bound to pass.

Spanner offers three main operation types: *read-write transactions*, *read-only transactions*, and *snapshot reads*. Read-write transactions require locks, pessimistic concurrency control, and presence of the leader replica. Read-only transactions are lock-free and can be executed at any replica. A leader is required only for reads at the *latest* timestamp, which takes the latest committed value from the Paxos group. Reads at the specific timestamp are consistent, since values are versioned and snapshot contents can't be changed once written. Each data record has a timestamp assigned,

which holds a value of the transaction commit time. This also implies that multiple timestamped versions of the record can be stored.

Figure 13-8 shows the Spanner architecture. Each *spanserver* (replica, a server instance that serves data to clients) holds several *tablets*, with Paxos (see Chapter 14) state machines attached to them. Replicas are grouped into replica sets called Paxos groups, a unit of data placement and replication. Each Paxos group has a long-lived leader (see Chapter 14). Leaders communicate with each other during multishard transactions.

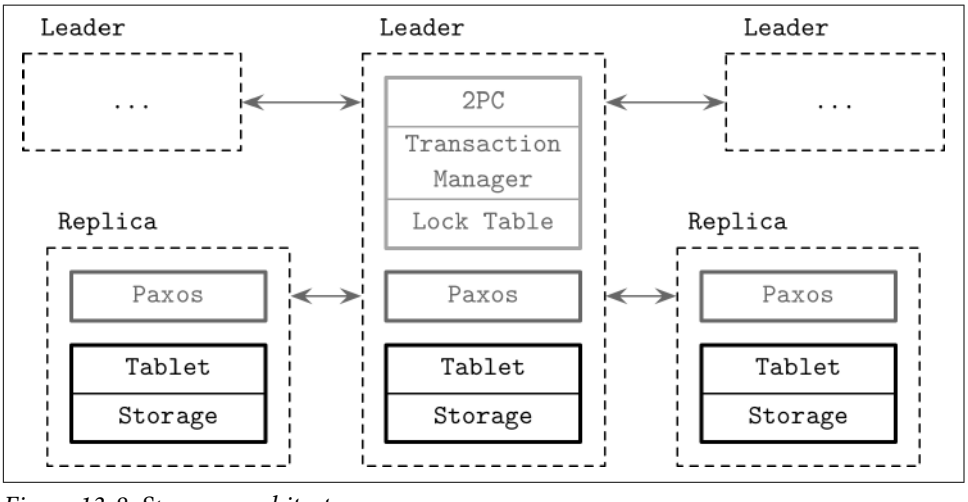


Figure 13-8. Spanner architecture

Every write has to go through the Paxos group leader, while reads can be served directly from the tablet on up-to-date replicas. The leader holds a *lock table* that is used to implement concurrency control using the two-phase locking (see Chapter 5) mechanism and a *transaction manager* that is responsible for multishard distributed transactions. Operations that require synchronization (such as writes and reads within a transaction) have to acquire the locks from the lock table, while other operations (snapshot reads) can access the data directly.

For multishard transactions, group leaders have to coordinate and perform a two-phase commit to ensure consistency, and use two-phase locking to ensure isolation. Since the 2PC algorithm requires the presence of all participants for a successful commit, it hurts availability. Spanner solves this by using Paxos groups rather than individual nodes as cohorts. This means that 2PC can continue operating even if some of the members of the group are down. Within the Paxos group, 2PC contacts only the node that serves as a leader.

Paxos groups are used to consistently replicate transaction manager states across multiple nodes. The Paxos leader first acquires write locks, and chooses a write time-

stamp that is guaranteed to be larger than any previous transactions' timestamp, and records a 2PC `prepare` entry through Paxos. The transaction coordinator collects timestamps and generates a commit timestamp that is greater than any of the prepare timestamps, and logs a `commit` entry through Paxos. It then waits until *after* the timestamp it has chosen for commit, since it has to guarantee that clients will only see transaction results whose timestamps are in the past. After that, it sends this timestamp to the client and leaders, which log the `commit` record with the new timestamp in their local Paxos group and are now free to release the locks.

Single-shard transactions do not have to consult the transaction manager (and, subsequently, do not have to perform a cross-partition two-phase commit), since consulting a Paxos group and a lock table is enough to guarantee transaction order and consistency within the shard.

Spanner read-write transactions offer a serialization order called *external consistency*: transaction timestamps reflect serialization order, even in cases of distributed transactions. External consistency has real-time properties equivalent to linearizability: if transaction T_1 commits before T_2 starts, T_1's timestamp is smaller than the timestamp of T_2.

To summarize, Spanner uses Paxos for consistent transaction log replication, two-phase commit for cross-shard transactions, and TrueTime for deterministic transaction ordering. This means that multipartition transactions have a higher cost due to an additional two-phase commit round, compared to Calvin [ABADI17]. Both approaches are important to understand since they allow us to perform transactions in partitioned distributes data stores.

Database Partitioning

While discussing Spanner and Calvin, we've been using the term *partitioning* quite heavily. Let's now discuss it in more detail. Since storing all database records on a single node is rather unrealistic for the majority of modern applications, many databases use partitioning: a logical division of data into smaller manageable segments.

The most straightforward way to partition data is by splitting it into ranges and allowing *replica sets* to manage only specific ranges (partitions). When executing queries, clients (or query coordinators) have to route requests based on the *routing key* to the correct replica set for both reads and writes. This partitioning scheme is typically called *sharding*: every replica set acts as a single source for a subset of data.

To use partitions most effectively, they have to be sized, taking the load and value distribution into consideration. This means that frequently accessed, read/write heavy ranges can be split into smaller partitions to spread the load between them. At the same time, if some value ranges are more dense than other ones, it might be a good idea to split them into smaller partitions as well. For example, if we pick *zip code* as a

routing key, since the country population is unevenly spread, some zip code ranges can have more data (e.g., people and orders) assigned to them.

When nodes are added to or removed from the cluster, the database has to repartition the data to maintain the balance. To ensure consistent movements, we should relocate the data before we update the cluster metadata and start routing requests to the new targets. Some databases perform *auto-sharding* and relocate the data using placement algorithms that determine optimal partitioning. These algorithms use information about read, write loads, and amounts of data in each shard.

To find a target node from the routing key, some database systems compute a *hash* of the key, and use some form of mapping from the hash value to the node ID. One of the advantages of using the hash functions for determining replica placement is that it can help to reduce range hot-spotting, since hash values do not sort the same way as the original values. While two lexicographically close routing keys would be placed at the same replica set, using hashed values would place them on different ones.

The most straightforward way to map hash values to node IDs is by taking a remainder of the division of the hash value by the size of the cluster (modulo). If we have `N` nodes in the system, the target node ID is picked by computing `hash(v) modulo N`. The main problem with this approach is that whenever nodes are added or removed and the cluster size changes from `N` to `N'`, many values returned by `hash(v) modulo N'` will differ from the original ones. This means that most of the data will have to be moved.

Consistent Hashing

In order to mitigate this problem, some databases, such as Apache Cassandra and Riak (among others), use a different partitioning scheme called *consistent hashing*. As previously mentioned, routing key values are hashed. Values returned by the hash function are mapped to a *ring*, so that after the largest possible value, it wraps around to its smallest value. Each node gets its own position on the ring and becomes responsible for the *range* of values, between its predecessor's and its own positions.

Using consistent hashing helps to reduce the number of relocations required for maintaining balance: a change in the ring affects only the *immediate neighbors* of the leaving or joining node, and not an entire cluster. The word *consistent* in the definition implies that, when the hash table is resized, if we have `K` possible hash keys and `n` nodes, on average we have to relocate only `K/n` keys. In other words, a consistent hash function output changes minimally as the function range changes [KARGER97].

Distributed Transactions with Percolator

Coming back to the subject of distributed transactions, isolation levels might be difficult to reason about because of the allowed read and write anomalies. If serializability

is not required by the application, one of the ways to avoid the write anomalies described in SQL-92 is to use a transactional model called *snapshot isolation* (SI).

Snapshot isolation guarantees that all reads made within the transaction are consistent with a snapshot of the database. The snapshot contains all values that were *committed before* the transaction's start timestamp. If there's a *write-write conflict* (i.e., when two concurrently running transactions attempt to make a write to the same cell), only one of them will commit. This characteristic is usually referred to as *first committer wins*.

Snapshot isolation prevents *read skew*, an anomaly permitted under the read-committed isolation level. For example, a sum of `x` and `y` is supposed to be `100`. Transaction `T1` performs an operation `read(x)`, and reads the value `70`. `T2` updates two values `write(x, 50)` and `write(y, 50)`, and commits. If `T1` attempts to run `read(y)`, and proceeds with transaction execution based on the value of `y` (`50`), newly committed by `T2`, it will lead to an inconsistency. The value of `x` that `T1` has read *before* `T2` committed and the new value of `y` aren't consistent with each other. Since snapshot isolation only makes values up to a specific timestamp visible for transactions, the new value of `y`, `50`, won't be visible to `T1` [BERENSON95].

Snapshot isolation has several convenient properties:

- It allows *only* repeatable reads of committed data.
- Values are consistent, as they're read from the snapshot at a specific timestamp.
- Conflicting writes are aborted and retried to prevent inconsistencies.

Despite that, histories under snapshot isolation are *not* serializable. Since only conflicting writes to the *same cells* are aborted, we can still end up with a *write skew* (see Chapter 5). Write skew occurs when two transactions modify disjoint sets of values, each preserving invariants for the data it writes. Both transactions are allowed to commit, but a combination of writes performed by these transactions may violate these invariants.

Snapshot isolation provides semantics that can be useful for many applications and has the major advantage of efficient reads, because no locks have to be acquired since snapshot data cannot be changed.

Percolator is a library that implements a transactional API on top of the distributed database *Bigtable* (see "Wide Column Stores" on page 15). This is a great example of building a transaction API on top of the existing system. Percolator stores data records, committed data point locations (write metadata), and locks in different columns. To avoid race conditions and reliably lock tables in a single RPC call, it uses a conditional mutation Bigtable API that allows it to perform read-modify-write operations with a single remote call.

Each transaction has to consult the *timestamp oracle* (a source of clusterwide-consistent monotonically increasing timestamps) twice: for a transaction start timestamp, and during commit. Writes are buffered and committed using a client-driven two-phase commit (see "Two-Phase Commit" on page 55).

Figure 13-9 shows how the contents of the table change during execution of the transaction steps:

- a) Initial state. After the execution of the previous transaction, `TS1` is the latest timestamp for both accounts. No locks are held.
- b) The first phase, called *prewrite*. The transaction attempts to acquire locks for all cells written during the transaction. One of the locks is marked as *primary* and is used for client recovery. The transaction checks for the possible conflicts: if any other transaction has already written any data with a later timestamp or there are unreleased locks at any timestamp. If any conflict is detected, the transaction aborts.
- c) If all locks were successfully acquired and the possibility of conflict is ruled out, the transaction can continue. During the second phase, the client releases its locks, starting with the primary one. It publishes its write by replacing the lock with a write record, updating write metadata with the timestamp of the latest data point.

Since the client may fail while trying to commit the transaction, we need to make sure that partial transactions are finalized or rolled back. If a later transaction encounters an incomplete state, it should attempt to release the primary lock and commit the transaction. If the primary lock is already released, transaction contents *have to be* committed. Only one transaction can hold a lock at a time and all state transitions are atomic, so situations in which two transactions attempt to perform operations on the contents are not possible.

		Data	Locks	Write Metadata
Account1	TS2	-	-	TS1 is latest
	TS1	$100	-	-
Account2	TS2	-	-	TS1 is latest
	TS1	$200	-	-

a) Initial State before moving $150 from Account2 to Account1

		Data	Locks	Write Metadata
Account1	TS3	$250	Primary	-
	TS2	-	-	TS1 is latest
	TS1	$100	-	-
Account2	TS3	$50	Primary at Account1	-
	TS2	-	-	TS1 is latest
	TS1	$200	-	-

b) State after taking locks and updating accounts

		Data	Locks	Write Metadata
Account1	TS4	-	-	TS3 is latest
	TS3	$250	-	-
	TS2	-	-	TS1 is latest
	TS1	$100	-	-
Account2	TS4	-	-	TS3 is latest
	TS3	$50	-	-
	TS2	-	-	TS1 is latest
	TS1	$200	-	-

c) Transaction commit releases locks and updates metadata with latest timestamp

Figure 13-9. Percolator transaction execution steps. Transaction credits $150 from Account2 and debits it to Account1.

Snapshot isolation is an important and useful abstraction, commonly used in transaction processing. Since it simplifies semantics, precludes some of the anomalies, and opens up an opportunity to improve concurrency and performance, many MVCC systems offer this isolation level.

One of the examples of databases based on the Percolator model is TiDB ("Ti" stands for Titatium). TiDB is a strongly consistent, highly available, and horizontally scalable open source database, compatible with MySQL.

Coordination Avoidance

One more example, discussing costs of serializability and attempting to reduce the amount of coordination while still providing strong consistency guarantees, is coordination avoidance [BAILIS14b]. Coordination can be avoided, while preserving data integrity constraints, if operations are invariant confluent. Invariant Confluence (*I*-Confluence) is defined as a property that ensures that two invariant-valid but diverged database states can be merged into a single valid, final state. Invariants in this case preserve consistency in ACID terms.

Because any two valid states can be merged into a valid state, *I*-Confluent operations can be executed without additional coordination, which significantly improves performance characteristics and scalability potential.

To preserve this invariant, in addition to defining an operation that brings our database to the new state, we have to define a *merge* function that accepts two states. This function is used in case states were updated independently and bring diverged states back to convergence.

Transactions are executed against the local database versions (snapshots). If a transaction requires any state from other partitions for execution, this state is made available for it locally. If a transaction commits, resulting changes made to the local snapshot are migrated and merged with the snapshots on the other nodes. A system model that allows coordination avoidance has to guarantee the following properties:

Global validity
: Required invariants are always satisfied, for both merged and divergent committed database states, and transactions cannot observe invalid states.

Availability
: If all nodes holding states are reachable by the client, the transaction has to reach a commit decision, or abort, if committing it would violate one of the transaction invariants.

Convergence
: Nodes can maintain their local states independently, but in the absence of further transactions and indefinite network partitions, they have to be able to reach the same state.

Coordination freedom
: Local transaction execution is independent from the operations against the local states performed on behalf of the other nodes.

One of the examples of implementing coordination avoidance is Read-Atomic Multi Partition (RAMP) transactions [BAILIS14c]. RAMP uses multiversion concurrency control and metadata of current in-flight operations to fetch any missing state

updates from other nodes, allowing read and write operations to be executed concurrently. For example, readers that overlap with some writer modifying the same entry can be detected and, if necessary, *repaired* by retrieving required information from the in-flight write metadata in an additional round of communication.

Using lock-based approaches in a distributed environment might be not the best idea, and instead of doing that, RAMP provides two properties:

Synchronization independence
: One client's transactions won't stall, abort, or force the other client's transactions to wait.

Partition independence
: Clients do not have to contact partitions whose values aren't involved in their transactions.

RAMP introduces the *read atomic* isolation level: transactions cannot observe any in-process state changes from in-flight, uncommitted, and aborted transactions. In other words, all (or none) transaction updates are visible to concurrent transactions. By that definition, the read atomic isolation level also precludes *fractured reads*: when a transaction observes only a subset of writes executed by some other transaction.

RAMP offers atomic write visibility without requiring mutual exclusion, which other solutions, such as distributed locks, often couple together. This means that transactions can proceed without stalling each other.

RAMP distributes transaction metadata that allows reads to detect concurrent in-flight writes. By using this metadata, transactions can detect the presence of newer record versions, find and fetch the latest ones, and operate on them. To avoid coordination, all local commit decisions must also be valid globally. In RAMP, this is solved by requiring that, by the time a write becomes visible in one partition, writes from the same transaction in all other involved partitions are also visible for readers in those partitions.

To allow readers and writers to proceed without blocking other concurrent readers and writers, while maintaining the read atomic isolation level both locally and system-wide (in all other partitions modified by the committing transaction), writes in RAMP are installed and made visible using two-phase commit:

Prepare
: The first phase prepares and places writes to their respective target partitions without making them visible.

Commit/abort
: The second phase publishes the state changes made by the write operation of the committing transaction, making them available atomically across all partitions, or rolls back the changes.

RAMP allows multiple versions of the same record to be present at any given moment: latest value, in-flight uncommitted changes, and stale versions, overwritten by later transactions. Stale versions have to be kept around only for in-progress read requests. As soon as all concurrent readers complete, stale values can be discarded.

Making distributed transactions performant and scalable is difficult because of the coordination overhead associated with preventing, detecting, and avoiding conflicts for the concurrent operations. The larger the system, or the more transactions it attempts to serve, the more overhead it incurs. The approaches described in this section attempt to reduce the amount of coordination by using invariants to determine where coordination can be avoided, and only paying the full price if it's absolutely necessary.

Summary

In this chapter, we discussed several ways of implementing distributed transactions. First, we discussed two atomic commitment algorithms: two- and three-phase commits. The big advantage of these algorithms is that they're easy to understand and implement, but have several shortcomings. In 2PC, a coordinator (or at least its substitute) has to be alive for the length of the commitment process, which significantly reduces availability. 3PC lifts this requirement for some cases, but is prone to split brain in case of network partition.

Distributed transactions in modern database systems are often implemented using consensus algorithms, which we're going to discuss in the next chapter. For example, both Calvin and Spanner, discussed in this chapter, use Paxos.

Consensus algorithms are more involved than atomic commit ones, but have much better fault-tolerance properties, and decouple decisions from their initiators and allow participants to decide on *a value* rather than on whether or not to accept *the value* [GRAY04].

Further Reading

If you'd like to learn more about the concepts mentioned in this chapter, you can refer to the following sources:

Atomic commitment integration with local transaction processing and recovery subsystems

Silberschatz, Abraham, Henry F. Korth, and S. Sudarshan. 2010. *Database Systems Concepts* (6th Ed.). New York: McGraw-Hill.

Garcia-Molina, Hector, Jeffrey D. Ullman, and Jennifer Widom. 2008. *Database Systems: The Complete Book* (2nd Ed.). Boston: Pearson.

Recent progress in the area of distributed transactions (ordered chronologically; this list is not intended to be exhaustive)

Cowling, James and Barbara Liskov. 2012. "Granola: low-overhead distributed transaction coordination." In *Proceedings of the 2012 USENIX conference on Annual Technical Conference (USENIX ATC '12)*: 21-21. USENIX.

Balakrishnan, Mahesh, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, Tao Zou, and Aviad Zuck. 2013. "Tango: distributed data structures over a shared log." In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*: 324-340.

Ding, Bailu, Lucja Kot, Alan Demers, and Johannes Gehrke. 2015. "Centiman: elastic, high performance optimistic concurrency control by watermarking." In *Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC '15)*: 262-275.

Dragojević, Aleksandar, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. "No compromises: distributed transactions with consistency, availability, and performance." In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*: 54-70.

Zhang, Irene, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. 2015. "Building consistent transactions with inconsistent replication." In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*: 263-278.

About the Author

Alex Petrov is a data infrastructure engineer, database and storage systems enthusiast, Apache Cassandra committer, and PMC member interested in storage, distributed systems, and algorithms.

Colophon

The animal on the cover of *Database Internals* is the peacock flounder, a name given to both *Bothus lunatus* and *Bothus mancus*, inhabitants of the shallow coastal waters of the mid-Atlantic and Indo-Pacific ocean, respectively.

While the blue floral-patterned skin gives the peacock flounder its moniker, these flounders have the ability to change their appearance based on their immediate surroundings. This camouflage ability may be related to the fish's vision, because it is unable to change its appearance if one of its eyes is covered.

Adult flatfishes swim in a horizontal attitude rather than in a vertical, back-up/belly-down, orientation as most other fishes do. When they swim, flatfishes tend to glide only an inch (2.54 cm) or so off the bottom while closely following the contour of the sea floor. One of this flat fish's eyes migrates during maturation to join the other on a single side, allowing the fish to look both forward and backward at once. Understandably, rather than swim vertically, the peacock flounder tends to glide an inch or so off the sea floor, closely following the contour of the terrain with its patterned side always facing up.

While the peacock flounder's current conservation status is designated as of Least Concern, many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on a black and white engraving from Lowry's *The Museum of Natural History*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.

SingleStore

A 5-Step Guide to Supercharging Your SaaS Apps

Powering Modern Data Applications with SingleStoreDB

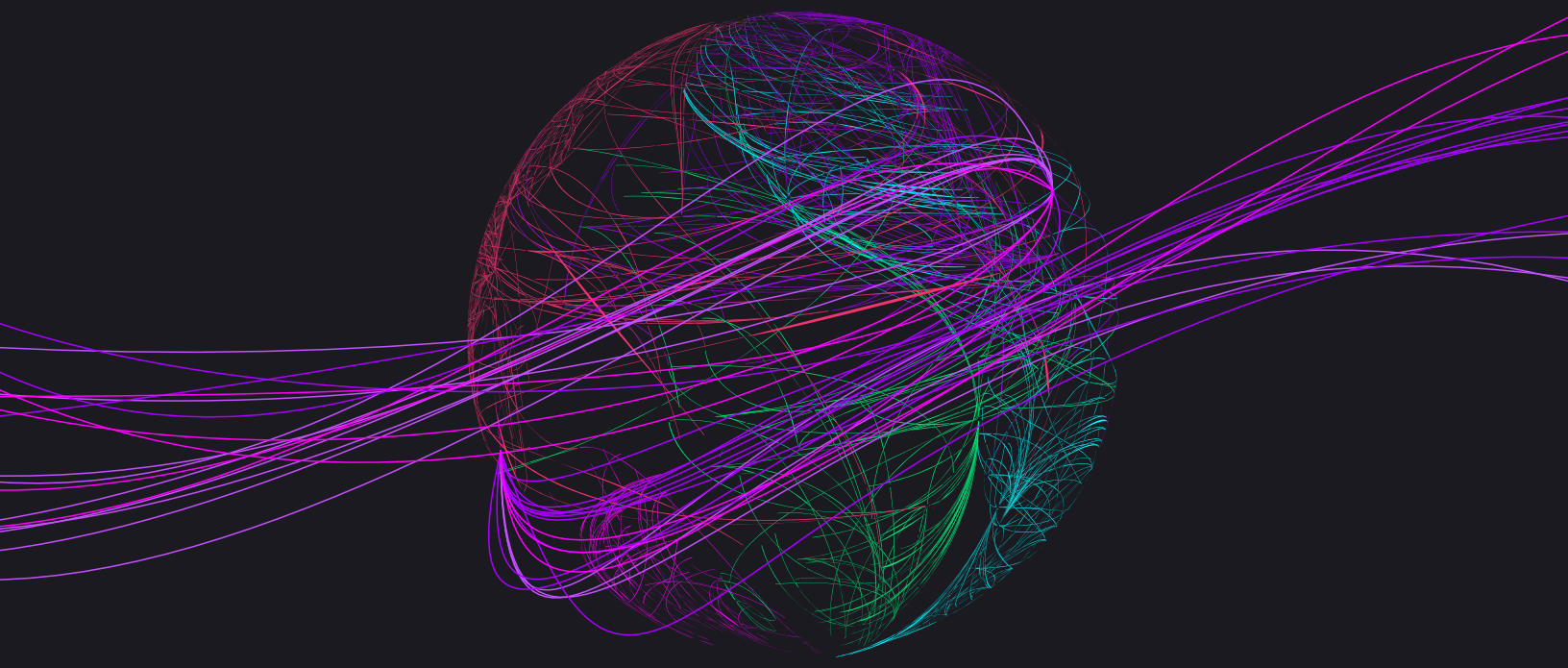


Table of Contents

Dear Fellow Developer and Database Technologist

When Tesla named their turbo mode "ludicrous" it sounded crazy — but people were more than pleasantly surprised to learn that it was indeed "ludicrously fast". Going from zero to 60 mph in 2.2 seconds literally takes your breath away, and leaves you in awe of the engineering marvel.

Wouldn't it be great to get that kind of customer reaction to your application experience?

Modern SaaS applications provide responsive, data-driven user experiences. They are cloud-native, distributed and composed of microservices and APIs. They are also data intensive in that the data processing of transactions and analytics is the gating factor, rather than being compute-bound or storage-bound, which the cloud has already solved. Modern SaaS apps are built to deliver real-time information and have the ability to scale to millions of users, on demand, everywhere. Whether users are choosing stocks or viewing the leaderboard, considering recommended content or redeeming points, those analytic queries present a constant challenge for Saas application developers to scale data infrastructure without slowing services or showing customers the dreaded spinning pinwheel.

Speed matters. In a competitive market, ludicrously fast customer experience (CX) is everything.

This eBook tells the story of three superstar application developers-Jack Ellis from Fathom Analytics,

Josh Blackburn of IEX Cloud and Gerry Morgan of DailyVest--who hit the accelerator on the analytics within their Saas products by improving the experience and speed by 50x and giving users the thrill of their own version of Ludicrous mode.

Read on to learn more about the 5 steps these developers took before choosing SingleStoreDB as the database engine to power their great customer experiences with real-time, interactive data analytics. Ludicrously fast.

Step 1: Scope the Problem
Step 2: Do Your Research
Step 3: Choose Wisely
Step 4: Talk With an Expert
Step 5: Migrate With Ease

Sincerely,
Domenic Ravita,
Field CTO

Meet the developers

Jack Ellis

fathom/

Jack is CTO and co-founder of Fathom Analytics, a SaaS firm that believes website analytics should be simple, fast and privacy focused. Fathom, now powered by SingleStoreDB, delivers a simple, lightweight, privacy-first alternative to Google Analytics. Jack chronicled his company's every step of their move from MySQL to SingleStoreDB in a detailed and very entertaining blog post, which also went viral in the developer community on Twitter. Here are a few highlights:

- First, there's the title of Jack's blog: "Building the world's fastest website analytics."
- The first sentence of the blog captures Fathom Analytics' enthusiasm for SingleStoreDB, too: "In March 2021, we moved all of our analytics data to the **database of our dreams.**"
- Of the SingleStoreDB sales process Jack said, "[T]his wasn't a sales call. This was a call where I could ask for help from engineers with 100x more knowledge than me, who have solved challenges for companies far larger than ours."

Josh Blackburn

iex cloud

Josh is co-founder and head of technology at IEX Cloud, a data infrastructure and delivery platform for financial and alternative data sets that connects developers and financial data creators. IEX Cloud is part of IEX Group, best known for the Investors Exchange. This US stock exchange was featured in Michael Lewis's book "Flash Boys", a literary bombshell that shook Wall Street.

Josh's team builds high-performance APIs and real time streaming data services used by hundreds of thousands of applications and developers-IEX Cloud has more than 130,000 users worldwide, and processes 1.2 billion API messages daily at 800,000 peak data operations per second.

In a webinar discussing why he chose SingleStoreDB Josh said, "The [SingleStoreDB] support for Apache Kafka has been phenomenal, especially as we are trying to **process hundreds of thousands of real-time prices.** That's just been an amazing feature. SingleStoreDB actually solved all of our problems for our use case, all in one database. It's very aptly named:'

Gerry Morgan

dailyVest

Gerry is lead developer at DailyVest, a fintech company using 401(k) participant data and analytics to improve the health and performance of retirement plans. Each month corporate clients upload investment and participant data to the cloud, where DailyVest assesses the "health" of each retirement plan; the firm analyzes the performance of $596 billion in assets, incorporating 3.3 billion transactions and the activity of 12.3 million anonymized participants. DailyVest then turns its customers' big data into digestible insights delivered via visual dashboards.

Sharing his experiences in a webinar with SingleStoreDB, Gerry said, "In initial benchmarking our stored procedures were up to three times faster, and we saw a **90% improvement in the time that it took to copy databases and restore them.** It was taking about an hour to do that in Azure SQL. The time is reduced to about four minutes in SingleStoreDB which, as far as we were concerned, was unbelievably good."

STEP 1

Scope the problem: A maxed-out database

They had all hit the wall.

Jack, Josh and Gerry all found SingleStoreDB through their individual quests to solve a common pain point: they had databases that could no longer keep up with the demands of their business.

What Fathom, IEX Cloud and dailyVest all have in common is that their applications qualify as data intensive — they need to offer fast, interactive experiences for thousands (or hundreds of thousands) of users on-demand, everywhere and in real time. And when it comes to powering these data-intensive applications, your underlying data engine makes all the difference.

Unfortunately, most organizations start with first-generation, open-source single-node databases to power their SaaS applications — quickly running into performance bottlenecks as analytics demands grow, or as applications need to scale. MySQL and PostgreSQL are among the most widely adopted and popular open-source databases on the planet, preferred by developers for how quickly, easily and cost-effectively they can get started. Unfortunately, these legacy, single-node architectures aren't built to handle analytics; as operations and data start to scale, their performance quickly deteriorates, leading to sluggish event-to-insight response times and rising costs. Even more, these databases aren't optimized to handle high throughput streaming ingestion, low-latency analytics and concurrency needs that digital disruptors — and their customers — demand.

Key Challenges With Legacy Gen-1 Data Engines

Streaming Ingestion

Inability to ingest, process and analyze streaming data necessary to power modern interactive SaaS applications

Low-Latency Query Performance

Lagging query performance as data or concurrency demands grow. Not optimized for low-latency queries

Challenges with Scaling

Built on single-node architectures and struggles to scale as your business or users grow

Minimal Analytical Capabilities

Offers little to no analytical capabilities to drive fast, interactive user experiences

fathom/

Fathom Analytics: Jack stumbled across SingleStoreDB on Twitter, where our Max Headroom ad had popped up in his feed. In his blog post Jack wondered: "What the hell does this even mean? Well, it's a play on a sci-fi TV show from the 80's called Max Headroom. I've never heard of it, my boomer friends, but it certainly made me click because, yes, I had indeed maxed out MySQL." Indeed, here's an example:

Despite keeping summary tables only (data rolled up by the hour), our [MySQL] database struggled to perform SUM and GROUP BY. And it was even worse with high cardinality data. One example was a customer who had 11,000,000 unique pages viewed on a single day. MySQL would take maybe 7 minutes to process a SUM/GROUP query for them, and our dashboard requests would just time-out. To work aroud this limitation, I had to build a dedicated cron job that pre-computed their dashboard data.

The first alternative Jack considered for scaling analytics for his app was Elasticsearch. While it might have solved part of the challenge, not having a standard SQL interface gave him pause. He writes, "This JSON approach and way of querying didn't feel good [in Elasticsearch]; high cardinality queries weren't performing as fast as I wanted, and I was sure I could get faster performance elsewhere,"

iex cloud

Josh had hit a similar wall with MySQL running in Google Cloud. He explained:

*We average about **500,000 to 800,000 data ops per second**, typically during market hours. These could be really tiny requests, but you can see our ingress and egress rates; we're consuming a lot of data from multiple resources, but we're also passing a lot of that out the door. In our case, we've got to keep up not just with the stock market, with real-time prices, but also with everyone coming in and needing all that data in real time.*

Josh summed up his challenge, "We were in a tight spot to find something that would scale and had better performance, especially on the ETL side , because we're loading hundreds of gigs of data every day."

dailyVest

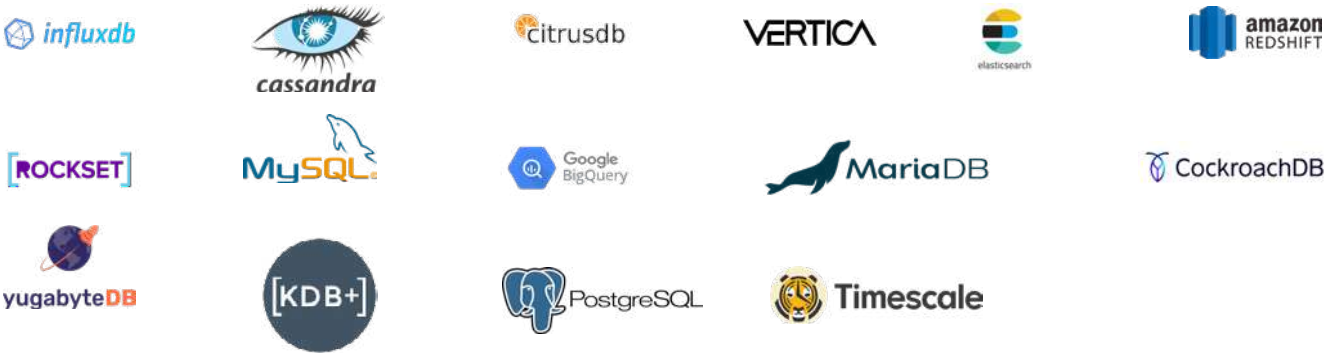
Data volumes are growing at 36% a year, fueled by billions of transactions. "What that meant for us was not just increasing resource requirements in our cloud environment, but also increasing costs [of Azure Cloud resources];' Gerry said. "What we were trying to do, in looking for a new database environment, was to maintain and even improve speed while reducing our monthly costs:' He added:

We were also seeing some performance degradation in Azure SQL. Not so much that our customers would have noticed, but we noticed there was some drop off in speed in our ingestion of data.We wanted to improve our ETL operation, but at the same time improve the customer experience-al/ customers will be happy if you make things faster, even if they haven't noticed if things were particularly slow.

90% PERFORMANCE IMPROVEMENT over Azure SQL with SingleStore

Do your research: Explore all the options

The developers chose SingleStoreDB after considering a multitude of alternative databases including:



Can we be friends?

The search for a "new friend" as Jack characterized it, included a lengthy list of non-negotiables.

- It must be ridiculously fast
- It must grow with us. We don't want to be doing another migration any time soon
- It must be a managed service. We are a small team, and if we start managing our database software, we've failed our customers. We're not database experts and would rather pay a premium price to have true professionals manage something as important as our customers' analytics data
- It must be highly available. Multi-AZ would be ideal, but high availability within a single availability zone is acceptable too
- Cost of ownership should be under $5,000/month. We didn't want to spend $5,000 off the mark, as this would be on top of our other AWS expenses, but we were prepared to pay for value
- The software must be mature
- Companies much larger than us must already be using it
- Support must be great
- Documentation must be well-written and easy to understand

Their needs differed

All three developers had specific requirements. For DailyVest, in addition to controlling costs, columnstore tables were a priority, in order to handle ad-hoc queries against large data volumes. IEX Cloud's data volumes demanded horizontal scalability, massive read and write speed, and support for bulk data loads.

Fathom Analytics' requirements went a layer deeper.

"For me;' Jack wrote, "I want the whole package. I like speed, but I also want to feel good about what I'm using. I want the people we're working with to be good people. And the technology has to fit into my existing knowledge in some way so that the learning curve isn't too large:'

STEP 3

Choose wisely: Recognize what matters most

As the developers put SingleStoreDB through its paces, each became more certain, and excited, about its potential to solve their challenges.

iex cloud

Perfect alignment.

For IEX Cloud, it quickly became clear that SingleStoreDB's capabilities aligned perfectly with the firm's needs. Josh said , "SingleStoreDB had all of the things we were looking for. And I'd been following SingleStoreDB for a long time:' He ran through the list of capabilities that won him over:

- From the very first call, I spoke with very knowledgeable people. [The SingleStoreDB sales engineer] was able to give recommendations, and **we were able to get SingleStoreDB up and running immediately because of its wire-support protocol for MySQL.**
- After only two years, IEX Cloud added **150,000 users in 120 countries, representing 20 million unique users downstream.**
- Ultimately, **choosing SingleStoreDB meant we didn't have months of migration time.** All the tools and support are already out there in the community.

Today, IEX Cloud **processes over 2.5B API requests daily,** with an 8ms average response time using SingleStoreDB — and a **10-15x increase in speed over their previous database.**

fathom/

Proof from peers.

Jack at Fathom Analytics was reassured by the success marquee-brand customers were already achieving with SingleStoreDB. He wrote:

[SingleStoreDB] gave specific use cases that made me confident they could handle us:

1. Comcast streaming **300,000 events per second**
2. Akamai handling **10,000,000 upserts per second**
3. A Tier-1 US bank handling real-time fraud protection with **50ms latency**

We are not even close to this level of scale. If these companies are using SingleStoreDB for that kind of scale, our use case should be a walk in the park.

With SingleStoreDB, Fathom ditched MySQL, Redis and DynamoDB to power their website analytics platform — **seeing a 1000x improvement in query performance and 60% reduction in database TCO.**

dailyVest

Requirements? Check.

DailyVest found that SingleStoreDB squarely met all of its requirements.

- Better performance: SingleStoreDB executes stored procedures up to three times faster than Azure SQL, and reduced copy-and-restore operations from one hour to four minutes, a **90% improvement.**
- Reduced TCO: SingleStoreDB total cost of ownership **saves DailyVest 35% over Azure SQL.**
- Hosted solution: Because it's a managed service, SingleStoreDB allows DailyVest to **avoid the expense and hassle of hosting a cluster in-house.**
- Stay in Azure Cloud: DailyVest could **switch to SingleStorDB from Azure SQL easily,** using existing client permissions for data storage in the Azure Cloud.

After moving to SingleStoreDB, dailyVest's batch process dropped from 4 hours and 12 minutes to **3 hours and 5 minutes,** resulting in **26.6% improvement.**

3X FASTER *SingleStoreDB executes stored procedures up to 3x faster than Azure SQL*

Choose wisely: Recognize what matters most

Break performance bottlenecks with SingleStoreDB

SingleStoreDB surpasses the limits of traditional data engines to drive up to 20-100x better performance, powering applications with analytics. SingleStoreDB offers a distributed, cloud-ready data platform with ANSI SQL compatibility that allows businesses to achieve fast ingest, ultra-fast query responses with high concurrency on real-time and historical data. SingleStoreDB supports analytics on streaming data by ingesting millions of rows per second on data-at-rest and data-in-motion. SingleStoreDB's architecture is designed to power data-intensive applications because of its unique ability to support both transactional and analytic workloads — all while enabling real-time analytics.

SingleStoreDB - Key Features

Patented Universal Storage: Both large-scale OLTP and OLAP are supported on this single, default table type. Universal Storage gives you the best qualities of row stores and column stores, while reducing data duplication, data movement and data latency.

SingleStoreDB Pipelines: Built-in parallel data ingestion technology natively ingests high-throughput, real-time data from external sources such as Apache Kafka, Amazon S3, Azure Blob, Filesystem, Google Cloud Storage and HDFS data source.

MySQL Compatibility: SingleStoreDB is wire-protocol compatible with MySQL/MariaDB which offers access to hundreds of languages, 100% compatibility on data types and 95% coverage of built-in functions to ease migrations.

Security & Compliance: Delivers enterprise-grade security with integrated user authentication, full encryption of data in transit and at rest, and SOC2, ISO27001, HIPAA, GDPR and CCPA compliance.

Separation of Storage and Compute for Transactions and Analytics: Allows users to effortlessly scale compute resources to meet the needs of any workload, while managing storage needs completely independently.

Distributed Ingest, Bulk or Streaming, with Lock-free/ Non-Blocking Reads and Concurrency: Offers a lock-free architecture that efficiently processes transactions and updates without locking or blocking concurrent reads — delivering the capability to perform bulk and/or streaming ingestion online, simultaneously with query workload.

Suspend & Resume Workloads Effortlessly: Clusters can be suspended and resumed nearly instantaneously, making all of your data available when you need it, minimizing costs when workloads are inactive.

Flexible Credit Pricing Model: Provides flexibility of on-demand, or with monthly credit bundles to handle dynamic and growing compute workloads at reduced TCO.

Latency-Free Analytics: SingleStoreDB lets you achieve ultra-fast query response with high concurrency across both live and historical data using familiar ANSI SQL.

Ultra-fast Event-to-insight Performance: Deliver against the toughest service-level agreements using parallel, distributed lock-free ingestion and real-time query processing.

Scale Limitlessly: Elastic scale-out architecture with distributed, massively parallel data processing delivers consistent, predictable responses under high ingest and user concurrency.

Ease of Use and Flexibility: SingleStoreDB brings simplicity and ease to your data processing by allowing transactional and analytical workloads to be processed using a single table type.

Tiered Storage: Three-tiered storage including in-memory, SSD Cache and the Cloud object store with separation of storage and compute (unlimited storage) and 80-90% data compression

Talk with an expert: A SingleStore engineer

Engineers trust engineers.

During the sales and implementation phases of their moves to SingleStoreDB, all of the developers were dazzled by the level of proactive, consultative help they received from SingleStoreDB's technical sales engineers and tech support team.

Superb technical support.

"We were very impressed with the technical support we were getting; it was really quite something;' said Gerry. During the early stages of DailyVest's implementation, he said, "we ran into a couple of problems which we didn't even notice ourselves. What we got was a proactive tech support call from SingleStoreDB:'

As it happens, DailyVest had many stored procedures that executed fastest in Azure SQL if the data was dumped into temp tables and later queried. Sometimes it's faster to use a common table expression (CTE)-in-memory queries that can then refer back to each other. "In SingleStoreDB you're always better off with CTEs" Gerry said. "The way I discovered this was from a call from SingleStoreDB, who said, "Hey, you're eating up most of the cluster with this temp table you keep creating and destroying.

That's a great way to do it in Azure SQL but not in SingleStoreDB: So we changed the code and now all of our stored procedures are CTE-based:'

Access to experts.

Jack at Fathom Analytics was impressed, too, with the access he had to SingleStoreDB experts. He wrote: "I fired off a few questions [to SingleStoreDB] a week or so after signing, and they came back with answers directly from a skilled engineer. The final cherry on top for me was when I sent them our schema. We had finalized it internally. We were going live in less than two weeks and needed an expert eye. Sarung Tripathi, [Principal Solutions Consultant at SingleStoreDB] checked it himself but also had their VP of Engineering [Robbie Walzer] look at it. Are you kidding me?"

"IEX Cloud got tremendous support, very knowledgeable technical support from the SingleStoreDB team;' Josh echoed. "We had our system, our unique build, up and running very quickly:'

Migrate with ease or don't migrate at all

Database migration is scary.

Why? Because it's risky. Migration weighs heavily on developers' minds as they evaluate new solut ions, because their business, reputation and sanity are all on the line.

Sometimes it just works.

Luckily, Josh sidestepped migration completely. SingleStoreDB is wire-compatible with MySQL, so IEX Cloud was able to "just get it up and running, and do a one-to-one comparison with what we already had in our system because it didn't require any code;' he said . " Ultimately, choosing [SingleStoreDB] meant that we didn't have months of migration time. There's a lot of tools and support already out there in the communit."

Sometimes it's a big deal.

Fathom Analytics' migration, on the other hand, was a 10-day marathon planned with a military level of precision. Jack detailed every step of the process in his blog, including code snippets. Here 's an abbreviated version of his migration story, which came off without a hitch:

This isn't my first rodeo. I've migrated countless high-value projects in the past. And even within Fathom, we've already done multiple migrations... But this migration was different because of the size of the data. We were dealing with hundreds of millions of rows, consisting of many billions of page views.

Many years ago, I read something from Tim Ferriss where he recommended imagining the worst possible case scenario for something, and then keep asking "and then." I use this technique for risk management in many areas of life and business, and I apply it to migrations too.

After finishing the migration, we were partying big time. This was months of work, doing research, implementations,and so much more. We couldn't believe we were finally migrated into a database system that could do everything we needed and was ready to grow with us. I spent the next few days watching the server metrics to ensure nothing would go wrong, and it was beautiful.

SingleStoreDB for Powering Your SaaS Applications

SingleStoreDB powers fast modern applications for over 100 leading SaaS players and Tier-1 enterprises around the globe. SingleStoreDB can effectively replace the need for multiple data engines to power your SaaS applications by enabling ultra-fast ingestion, super low-latency queries and multi-model support with unlimited storage. Customers can deploy SingleStoreDB in any of the leading cloud environments including AWS, Azure, GCP, or in a hybrid mode.

Key Attributes for Modern SaaS Apps	SingleStoreDB - Key Capabilities
Power Fast & Interactive Data Experiences	Distributed SQL platform bringing together fast transactions and analytics in the same engine in real time, with no data movement. See recent TPC Benchmarking results
Access to Real-Time Data	Parallel streaming ingestion up to millions of events/second using Pipelines, together with super-low latency queries
Scale Effortlessly	Infinite elasticity to scale your applications including scale-out HTAP, with separation of storage and compute (Unlimited Data Storage)
Handle Any Data, Run Anywhere	Multi-model; support multiple data types (JSON, time-series, geo, full-text search, relational) and run in multi-cloud or hybrid-clouds
Resiliency & Recoverability	Best-in-class resiliency to power enterprise apps including High Availability, Disaster Recovery, Limitless PITR and Multi-AZ failover
Frictionless Developer Experience	Simplicity and programmability including MySQL wire protocol and connectors to Spark, Kafka and dbt to quickly launch new apps

The Results: Faster, better, cheaper

Operation	Azure SQL vs SingleStoreDB		Latency Reduction	Performance Improvement
Generate cached financials, 1 plan, 12 months	59 mins, 27 s	3 mins, 15 s	-94.5%	18.3x faster
Data replication	60 mins	4 mins	-93.3%	15x faster
Sum of all transactions and group by month	64.3 s	47.9 s	-25.5%	1.3x faster
Count all participants and group by month	1062 ms	930 ms	-12.4%	1.15x faster
Generate all KPMs for latest month	4 hrs, 12 mins	3 hrs, 5 mins	-26.6%	1.3x faster

Gerry at DailyVest provided this summary table of his testing results.

Real Benefits. No Lie.

Fathom Analytics, IEX Cloud and DailyVest all have benefited from the performance improvements gained from SingleStoreDB; Gerry at DailyVest provided the summary table above, while Jack dove deep into details:

1. *We no longer need a dedicated data-export environment...We do our data exports by hitting SingleStoreDB with a query that it will output to 53 for you typically within less than 30 seconds. It's incredible. This means we can export gigantic (lies to 53 with zero concern about memory. We would regularly run into data export errors for our bigger customers in the past, and I've spent many hours doing manual data exports for them. I cannot believe that is behind me. I'm tearing up just thinking about it.*

2. *Our queries are unbelievably fast. A day after migrating, two of my friends reached out telling me how insanely fast Fathom was now, and we've had so much good feedback.*

3. *We can update and delete hundreds of millions of rows in a single query. Previously, when we needed to delete a significant amount of data, we had to chunk up deletes into DELETE with LIMIT. But SingleStoreDB doesn't need a limit and handles it so nicely*

4. *We used to have a backlog, as we used INSERT ON DUPLICATE KEY UPDATE for our summary tables...*

[W]e had to put sites into groups to run multiple cron jobs side by side, aggregating the data in isolated (by group) processes. But guess what? Cron jobs don't scale, and we were starting to see bigger pageview backlogs each day. Well, now we're in SingleStoreDB, data is fully real-time. So if you view a page on your website, it will appear in your Fathom dashboard with zero delays.

5. *Our new database is sharded and can filter across any (,eld we desire. This will support our brand new, Version 3 interface, which allows filtering over EVERYTHING.*

6. *We are working with a team that supports us. I often feel like I'm being cheeky with my questions, but they're always so happy to help. We're excited about this relationship.*

7. *SingleStoreDB has plans up to $119,000/month, which is hilarious. That plan comes with 5TB of RAM and 640 vCPU. I don't think we'll get there any time soon, but it feels good to see they're comfortable supporting that kind of scale. They're an exciting company because they're seemingly targeting smaller companies like us, but they're ready to handle enterprise-scale too.*

8. And as for price, we're spending under $2,000/month, and we're over-provisioned, running at around 10% - 20% CPU most of the day.

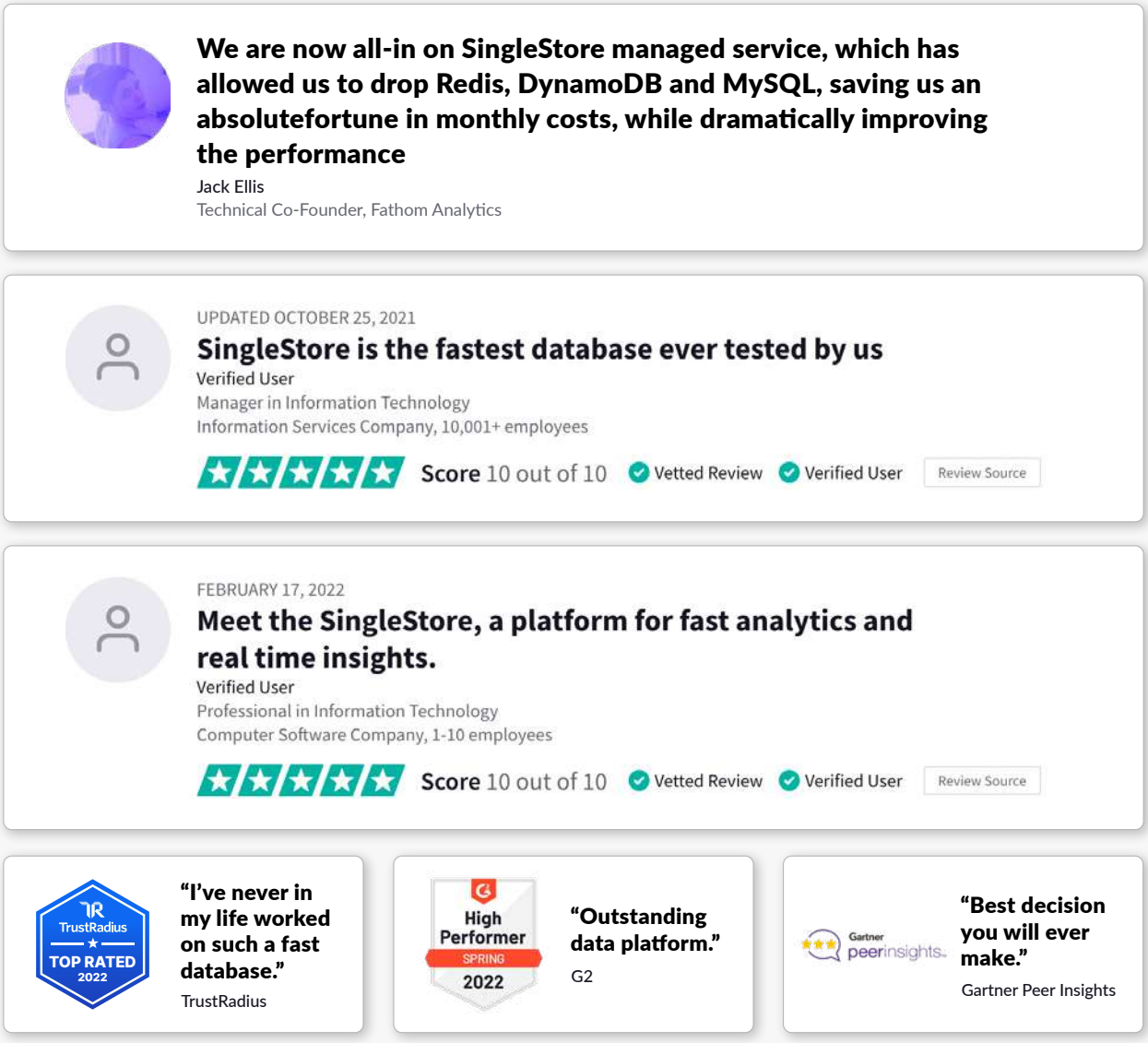
The Results: Faster, better, cheaper

It's not too good to be true.

Any application developer will tell you it's true: poor performance of in-app analytics translates into a poor customer experience, which is a direct threat to reputation and revenues.

Enormous efficiencies-all for you.

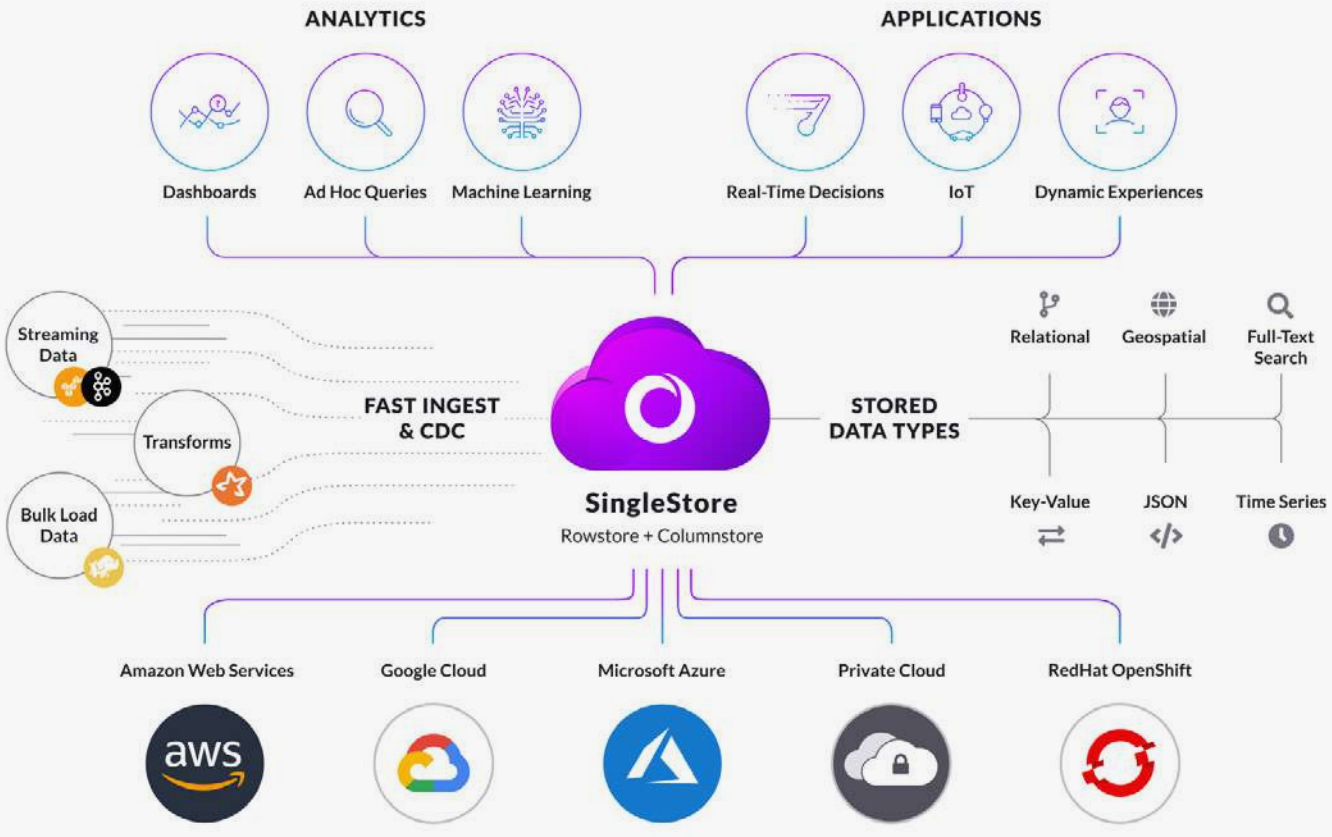
Josh summed up, "SingleStoreDB enables us to do monitoring and analysis in the same system that houses our historical data, and this creates enormous efficiencies for us. We've been able to consolidate multiple databases, run our platform faster, and speed the onboarding processes for new data sets."



About SingleStore: Ludicrously fast analytics

We've got you.

SingleStoreDB is the cloud-native database built with the speed and scale to power the world's data-intensive applications. With a distributed SQL database that unifies transactions and analytics, SingleStoreDB empowers digital giants like Uber, Hulu and Comcast to deliver memorable, limitless data experiences. Built to handle various data types, SingleStoreDB supports multiple data types (including JSON, time-series, geospatial and full-text search), and runs in multi-cloud environments.



Dramatic improvements.

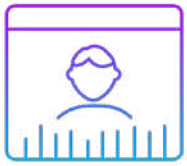
Fathom Analytics, IEX Cloud and DailyVest all have dramatically improved their applications' performance with SingleStoreDB's best-in-class speed, scale and capability, without the headaches of installing, configuring and maintaining software. To get started today, for free, visit www.singlestore.com/try-free.

About SingleStore: Who we are

SingleStoreDB delivers the cloud-native database with the speed and scale to power the world's data-intensive applications. With a distributed SQL database that introduces simplicity to your data architecture by unifying transactions and analytics, SingleStoreDB empowers digital leaders to deliver exceptional, real-time data experiences to their customers.

Used by the Most Innovative Companies In the World

Half of the Top 10 Banks



Streaming Media Leaders in Music, Video & Gaming

12 of the Fortune 50

Tech Innovators from Akamai and Uber

2 of the Top 3 Telcos

Experience SingleStore for yourself!

Install the SingleStoreDB for FREE or Deploy our managed Service with $500 in FREE credits.

SingleStore

For more details visit us at SingleStore.com