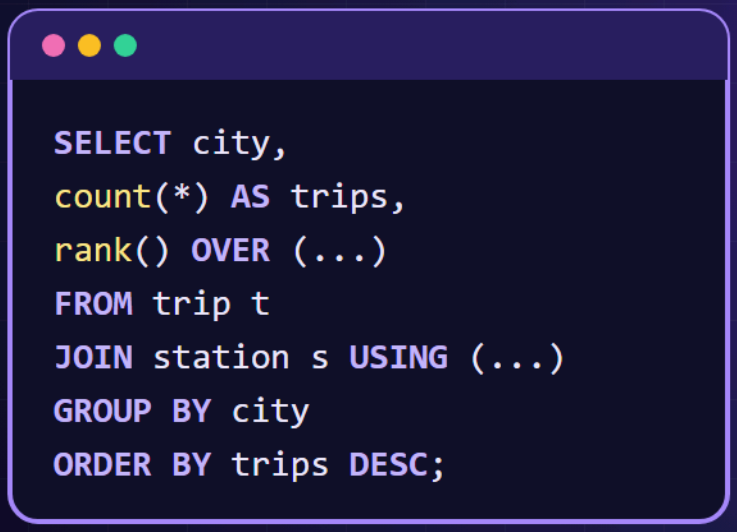
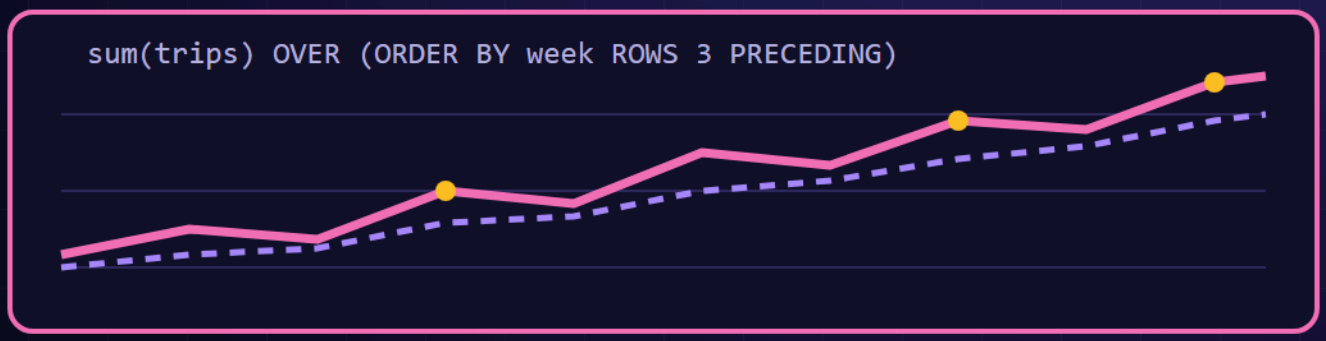


city	trips
north	
harbor	
east	
river	



JOIN · GROUP BY · WINDOW · CTE · EXPLAIN · INDEX

FROM ZERO TO ADVANCED

SQL for Developers and Data Analysts

Queries, analysis, data design, and performance tuning

With PostgreSQL, plus notes for MySQL, SQL Server, SQLite, and cloud data warehouses

SELECT · JOINS · WINDOWS · INDEXES · EXPLAIN

THENEXBOOKS

SQL for Developers and Data Analysts

From Zero to Advanced Queries and Performance Tuning
With PostgreSQL, plus notes for MySQL, SQL Server, SQLite, and cloud warehouses

First Edition

THENEXBOOKS

SQL for Developers and Data Analysts
From Zero to Advanced Queries and Performance Tuning

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. Database systems evolve quickly; always consult the official documentation for the product and version you use. Execution plans, costs, and timings shown in this book are illustrative — your results will differ.

PostgreSQL is a trademark or registered trademark of the PostgreSQL Community Association of Canada. MySQL is a trademark of Oracle and/or its affiliates. Microsoft and SQL Server are trademarks of the Microsoft group of companies. Other product names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement. This book is not affiliated with, sponsored by, or endorsed by any of these organizations.

Spokeway, the running example in this book, is fictional, as are all members, stations, and companies mentioned in its examples. Any resemblance to real companies or products is coincidental.

Preface

SQL is one of the rare technologies that has outlived nearly every trend around it. Programming languages come and go, frameworks rise and fall, and yet the question "what does the data actually say?" is still answered, overwhelmingly, with a query. Application developers write SQL to store and fetch the data their software runs on. Data analysts write SQL to turn that same data into answers, dashboards, and decisions. Increasingly, the same person does both.

Learning the basics of SQL takes an afternoon. Becoming genuinely good at it takes much longer — and most of the difficulty hides in places beginners never see coming: `NULL` quietly changing the meaning of a filter, a join that silently multiplies rows, an average that is wrong because of how it was grouped, a report that shifts by a day because of time zones, and a query that was instant in development and takes minutes in production.

This book takes you from your first `SELECT` to advanced analytical queries and real performance tuning. We explain *why* each technique works and where it breaks, not just *how* to type it, and we use one realistic database throughout so every new idea builds on the last.

Who This Book Is For

- **Software developers** who use SQL from application code and want correct results, sound schemas, safe changes, and fast queries.
- **Data analysts** who query databases and warehouses and want to move from basic reports to confident, advanced analysis.
- **Beginners** starting from zero — Part I assumes no prior SQL.
- **Experienced users** who want to master window functions, analytical patterns, execution plans, and indexing.

What You Will Learn

- The relational model, data types, `NULL` and three-valued logic, and the core `SELECT` statement.
- Aggregation, every kind of join, subqueries, common table expressions, recursive queries, set operations, and pivoting.
- Window functions for ranking, running totals, moving averages, and period-over-period comparisons.
- Time-series analysis, cohorts, retention, funnels, sessions, and gaps-and-islands.
- JSON, arrays, and full-text search in a relational database.
- Data modeling, DDL and constraints, safe data changes, upserts and `MERGE`, transactions, and concurrency.
- Views, materialized views, functions, and procedures.
- Reading execution plans, designing indexes, tuning slow queries, and managing large tables.

- SQL from application code, differences between major databases, and a complete case study.

How This Book Is Organized

Part I — SQL Foundations (Chapters 1–5) sets up your environment and teaches the core of querying: tables, filtering, sorting, data types, `NULL`, and functions. **Part II — Combining and Summarizing Data** (Chapters 6–10) covers aggregation, joins, subqueries, CTEs, set operations, and pivots. **Part III — Advanced Querying for Analysis** (Chapters 11–15) is where SQL becomes an analysis language: window functions, time series, analytical patterns, and semi-structured data. **Part IV — Designing and Changing Data** (Chapters 16–20) covers modeling, DDL, data changes, transactions, and database objects. **Part V — Performance Tuning** (Chapters 21–25) explains how queries run and how to make them fast. **Part VI — SQL in Practice** (Chapters 26–28) connects SQL to applications and other databases, and finishes with a complete case study. The appendices provide a cheat sheet, templates and a function reference, a production checklist, and a glossary.

Each chapter includes practice exercises and ends with a **Key Takeaways** box for quick review.

The Running Example: Spokeway

Throughout the book we work with **Spokeway**, a fictional bike-sharing network operating in several cities. Members sign up for plans, unlock bikes at stations, ride, and pay; stations report how many bikes and docks are free; bikes break and get repaired; and the mobile app records what riders do. That gives us exactly the mix of data both developers and analysts meet every day: transactional records, time-stamped events, money, and messy real-world edge cases. Every chapter queries the same `spokeway` database, so you can follow along and build on what you wrote before. Chapter 2 creates the schema and a small set of starter rows; many later chapters insert a few extra, clearly numbered sample rows of their own to set up the exact situation being discussed — a guest ride, a refunded payment, a station with no trips — and the results they show are based on those rows.

Versions and Conventions

Examples use PostgreSQL, recent versions (16 and later), because it is free, runs everywhere, and follows the SQL standard closely. Where a feature needs a specific version, we say so. Short *dialect notes* show how to do the same thing in MySQL, SQL Server, SQLite, and cloud data warehouses, and Chapter 27 compares them in depth.

> **TIP**
> A recommended practice or a technique that makes your work easier.

> **NOTE**
> Background information, version details, dialect differences, or a useful aside.

> **WARNING**
>
> A common mistake or a behavior that can cause wrong results, data loss, or slow queries in production.

SQL keywords are written in uppercase and identifiers in lowercase `snake_case`. Query results are shown in a compact, psql-style layout. Execution plans and timings are illustrative.

Contents

Preface

PART I — SQL FOUNDATIONS

1 Why SQL Still Matters: Databases, Tables, and the Relational Model

2 Setting Up: PostgreSQL, the Spokeway Sample Database, and Your SQL Toolkit

3 Your First Queries: SELECT, WHERE, ORDER BY, and LIMIT

4 Data Types and NULL: Numbers, Text, Dates and Times, Booleans, and Three-Valued Logic

5 Expressions and Functions: CASE, String, Numeric, and Date/Time Functions

PART II — COMBINING AND SUMMARIZING DATA

6 Aggregation: GROUP BY, HAVING, and Summary Statistics

7 Joins in Depth: INNER, OUTER, CROSS, Self Joins, and Join Pitfalls

8 Subqueries: Scalar, Correlated, IN, and EXISTS

9 Common Table Expressions and Recursive Queries

10 Set Operations, Conditional Aggregation, and Pivoting

PART III — ADVANCED QUERYING FOR ANALYSIS

11 Window Functions I: Ranking, Numbering, and Top-N per Group

12 Window Functions II: Running Totals, Moving Averages, LAG/LEAD, and Frames

13 Dates and Time Series: Bucketing, Time Zones, Gaps, and Calendars

14 Analytical Patterns: Cohorts, Retention, Funnels, Sessions, and Gaps-and-Islands

15 Semi-Structured Data: JSON, Arrays, and Full-Text Search

PART IV — DESIGNING AND CHANGING DATA

16 Data Modeling and Normalization for Developers and Analysts

17 Creating Tables: DDL, Keys, Constraints, Defaults, and Generated Columns

18 Changing Data: INSERT, UPDATE, DELETE, Upserts, and MERGE

19 Transactions, Isolation Levels, Locking, and Concurrency

20 Views, Materialized Views, Functions, and Stored Procedures

PART V — PERFORMANCE TUNING

21 How Queries Run: The Planner, EXPLAIN, and EXPLAIN ANALYZE

22 Indexing Fundamentals: B-Tree Indexes and How the Planner Uses Them

23 Advanced Indexing: Composite, Covering, Partial, Expression, GIN, GiST, and BRIN

24 Query Tuning Patterns: Diagnosing and Rewriting Slow Queries

25 Large Tables: Statistics, Vacuum, Partitioning, and Batch Operations

PART VI — SQL IN PRACTICE

26 SQL from Application Code: Parameters, Injection, ORMs, N+1, and Pagination

27 SQL Across Databases: MySQL, SQL Server, SQLite, and Cloud Data Warehouses

28 Case Study: Building the Spokeway Operations and Analytics Database

APPENDICES

A SQL Cheat Sheet

B Templates and Function Reference

C Production Checklist

D Glossary

PART I

SQL Foundations

What relational databases are and why SQL still matters, a working PostgreSQL setup with the Spokeway database, and the core of querying: selecting, filtering, sorting, data types, NULL, and functions.

1 · Why SQL Still Matters
2 · Setting Up
3 · Your First Queries
4 · Data Types and NULL
5 · Expressions and Functions

CHAPTER 1 · PART I — SQL FOUNDATIONS

Why SQL Still Matters: Databases, Tables, and the Relational Model

Before writing a single query, it pays to understand what a relational database actually is, why SQL has outlived generations of programming fashions, and how tables, rows, keys, and relationships fit together. This chapter builds that mental model using Spokeway, the bike-sharing network you will query throughout the book.

Two Readers, One Language

This book is written for two groups of people who often sit in the same company but rarely read the same books. The first group is software developers: people who build web services, mobile back ends, batch jobs, and internal tools, and who send SQL to a database from application code. The second group is data analysts: people who answer questions such as "How many members did we keep after three months?" or "Which stations run out of bikes every weekday morning?" by querying databases and warehouses directly.

Both groups use the same language, but they use it differently. A developer typically writes a small number of queries that run thousands or millions of times a day, so correctness under concurrency and predictable performance matter enormously. An analyst typically writes many queries that run a handful of times, so expressiveness, readability, and getting the *right* answer from messy data matter most. The good news is that the underlying skills overlap almost completely. A developer who understands window functions writes better reporting endpoints; an analyst who understands indexes and query plans stops waiting twenty minutes for a dashboard to refresh.

When a topic matters more to one group, we say so directly ("From application code…", "If you are an analyst…"). Read both — the best SQL users think like both.

What a Database Is — and What It Is Not

A **database** is an organized collection of data that is stored so it can be retrieved, changed, and protected reliably. That definition is broad enough to include a spreadsheet, a folder of CSV files, or a JSON document on disk. What turns a pile of data into something you can trust for a business is the software that manages it: the **database management system**, or DBMS.

A DBMS takes responsibility for a set of problems that are genuinely hard to solve correctly by hand:

- **Durability.** Once the database says a change is saved, that change survives a crash or power loss.
- **Concurrency.** Many users and programs can read and write at the same time without corrupting each other's work.

- **Integrity.** Rules such as "every trip must reference a real bike" are enforced for every change, not just the ones a careful programmer remembered to check.
- **Access control.** Different users can be allowed to see or change different data.
- **Efficient retrieval.** You describe the data you want; the system works out how to find it, using indexes and statistics you may never see.

A spreadsheet provides almost none of these guarantees. Two people editing the same file overwrite each other, nothing stops a typo in a numeric column, and "find all rows where…" becomes painfully slow once the data grows. That is not a criticism of spreadsheets — they are superb for exploration — but it explains why nearly every serious application eventually stores its data in a real database.

Relational and non-relational systems

There are many kinds of DBMS. **Relational** databases organize data into tables with a fixed set of columns and use SQL as their language. PostgreSQL, MySQL, SQL Server, Oracle Database, and SQLite are all relational. **Non-relational** systems — often grouped under the label "NoSQL" — include document stores, key-value stores, wide-column stores, and graph databases. Each makes different trade-offs, and each is a good fit for certain workloads.

Over the years, the boundaries have blurred. Relational databases added support for JSON documents (Chapter 15 covers PostgreSQL's `jsonb` type in depth), and several non-relational systems added SQL-like query languages because their users kept asking for them. Cloud data warehouses, which analysts use heavily, are relational at heart: they store tables and speak SQL, even though their internal storage is organized very differently from a transactional database. Learning SQL well therefore pays off across a surprisingly wide range of products.

Why SQL Has Lasted

SQL was designed in the 1970s, and technologies that old are usually museum pieces. SQL is not, and the reasons are worth understanding because they shape how you should write it.

SQL is declarative

Most programming languages are **imperative**: you tell the computer *how* to do something, step by step. SQL is **declarative**: you describe *what* result you want, and the database decides how to produce it. Compare the two approaches to finding the active stations in one city.

imperative-pseudocode.txt

```
result = []
for each row in station_file:
    if row.city_id == 1 and row.is_active:
        result.append(row.name)
sort result alphabetically
return result
```

declarative.sql

```
SELECT name
FROM station
WHERE city_id = 1
  AND is_active
ORDER BY name;
```

The SQL version says nothing about loops, files, or sort algorithms. That omission is the whole point. If the table holds a few dozen rows, the database may simply read them all. If it holds millions and there is an index on `city_id`, the database may jump straight to the matching rows. If the data is spread across many machines, a distributed engine may split the work. Your query does not change; the **query planner** chooses a strategy based on the data it has. Chapter 21 shows you how to see those choices with `EXPLAIN`.

The practical consequence is that SQL code often outlives the hardware and data volumes it was written for. A report written when the company had one city can keep working, unchanged, when it has twenty — and if it gets slow, the fix is frequently an index or better statistics rather than a rewrite.

SQL is set-based

SQL operates on whole sets of rows at a time. A single `UPDATE` statement can change one row or ten million; a single `SELECT` can combine several tables and summarize the result. This is a different habit of mind from row-by-row programming, and it is the habit that separates comfortable SQL users from struggling ones. Developers coming from loops-and-objects languages often write SQL that fetches rows one at a time and processes them in application code. It works, but it is usually slower and harder to keep correct than letting the database do the work in one statement. Chapter 26 returns to this theme when we discuss the "N+1 query" problem.

SQL is standardized — mostly

SQL is defined by an international standard that has been revised many times. Every major product implements a large common core of it, plus its own extensions. That core — `SELECT`, `WHERE`, `JOIN`, `GROUP BY`, `INSERT`, `UPDATE`, `DELETE`, and much more — transfers between products with little or no change. The differences tend to appear at the edges: date functions, string concatenation, how you limit the number of rows returned, and procedural extensions. Chapter 27 walks through the most important differences between PostgreSQL, MySQL, SQL Server, SQLite, and cloud warehouses.

> **NOTE**
>
> This book uses PostgreSQL as its primary dialect, targeting recent versions (16 and later). PostgreSQL is free, open source, runs on every major operating system, and follows the SQL standard closely in most areas. When a feature requires a specific version, or behaves differently elsewhere, we point that out in a short "Dialect note".

Meet Spokeway

Every example in this book uses the same fictional company, so that the data becomes familiar and you can focus on the SQL rather than on a new domain each chapter. **Spokeway** is a city bike-sharing network operating in several cities. Riders pick up a bike at a docking station, ride it, and return it to any other station. Some riders are *members* on a plan; others are *guest riders* who pay per trip without an account.

Spokeway's data naturally raises the kinds of questions both our audiences face:

- The mobile app must show, in real time, how many bikes are available at a station (a developer's problem: fast, correct lookups under constant updates).
- Operations staff want to know which stations run empty during the morning commute so they can rebalance bikes (an analyst's problem: time-series analysis).
- Finance wants revenue per plan per month, including refunds (aggregation and money handling).
- Product managers want to know how many members who signed up in March were still riding in June (cohort retention).
- Maintenance wants to find bikes that keep coming back for repairs (grouping and filtering over workflow data).

The Spokeway tables

The Spokeway database, called `spokeway`, contains ten core tables. You will create them in Chapter 2; for now, here is what each one holds.

Table	One row per…	Example question
`city`	city Spokeway operates in	Which cities are in which country?
`station`	docking station	Which stations have more than 20 docks?
`bike`	physical bike	How many electric bikes are in maintenance?
`plan`	membership plan	What does the Annual plan cost?
`member`	registered rider	Who signed up this month?
`trip`	ride from one station to another	What is the median trip length?
`payment`	charge, refund, or failure	What was revenue last week?
`maintenance_ticket`	repair job on a bike	Which bikes had 3+ tickets?
`station_status`	periodic station snapshot	When did station 7 run empty?
`app_event`	event in the mobile app	How many opens led to a trip?

Notice that each table describes one kind of thing, and that the answer to "one row per..." is always a single, clear phrase. That is not an accident; it is the first principle of good relational design, and we will lean on it constantly. When you meet an unfamiliar table at work, asking "what does one row represent?" is the fastest way to understand it — and the most common source of wrong answers when you guess incorrectly.

Tables, Rows, and Columns

A **table** is a named collection of rows that all share the same columns. Each **column** has a name and a **data type** such as integer, text, date, or timestamp. Each **row** holds one value (or a missing value, `NULL`) for every column. Here is a small slice of the `station` table as `psql`, PostgreSQL's command-line client, would display it.

```
 station_id | city_id |       name        | capacity | is_active
------------+---------+-------------------+----------+-----------
          1 |       1 | Harbor Square     |       24 | t
          2 |       1 | Old Mill Bridge   |       16 | t
          3 |       1 | University Gate   |       30 | t
          4 |       2 | Central Market    |       20 | t
          5 |       2 | Riverside Park    |       12 | f
(5 rows)
```

Some vocabulary is worth fixing now, because you will meet it in documentation and error messages. In the formal relational model, a table is a **relation**, a row is a **tuple**, and a column is an **attribute**. Older texts sometimes say "record" for a row and "field" for a column. This book uses the everyday terms — table, row, column — but you should recognize the others.

Rows have no inherent order

One of the most important facts about relational tables surprises almost everyone at first: **rows in a table have no defined order**. The output above happens to be sorted by `station_id`, but nothing guarantees that. If you run the same query twice, you will very often get the same order, because the database reads the data the same way. Then one day a table grows, or a row is updated and moves to a different place on disk, or the planner picks a parallel scan, and the order changes.

If order matters, you must ask for it explicitly with `ORDER BY`, which Chapter 3 covers. This single rule prevents a whole class of production bugs, such as a paginated API that sometimes shows the same item on two pages, or a report whose "first row" silently changes.

> **WARNING**
>
> Never rely on the order in which rows "usually" come back. Without `ORDER BY`, the database is free to return rows in any order, and that order can change after an upgrade, a data change, or a different execution plan. Code and reports that assume an order work perfectly in testing and then fail unpredictably in production.

Columns have types

Unlike a spreadsheet cell, a column in a relational table accepts only values of its declared type. If `capacity` is an integer column, the database rejects `'twenty'`. This strictness feels restrictive when you are exploring, but it is precisely what lets everyone downstream trust the data: a sum over `capacity` will never be derailed by a stray word. Chapter 4 covers PostgreSQL's data types in detail, including the subtle ones — money, time zones, and `NULL`.

Keys: How Rows Are Identified

If rows have no order, how do you refer to a particular one? You use a **key**: a column (or combination of columns) whose value identifies exactly one row.

Primary keys

Every well-designed table has a **primary key**. In `station`, the primary key is `station_id`. The database enforces two rules for a primary key: its value must be unique across the table, and it must not be `NULL`. Try to insert a second station with `station_id = 1` and PostgreSQL refuses with an error.

Spokeway uses **surrogate keys**: numeric identifiers with no meaning outside the database. The alternative is a **natural key**, a value that already identifies the thing in the real world, such as a country code or an email address. Natural keys look attractive, but real-world identifiers change more often than you expect — people change email addresses, stations get renamed — and changing a primary key means changing every reference to it. Surrogate keys avoid that problem. Chapter 16 discusses the trade-offs more fully.

That does not mean natural identifiers are unimportant. Spokeway's `member.email` is not the primary key, but it is declared `UNIQUE`, so two members can never share an email address. A unique constraint like this is sometimes called an **alternate key**: the database guarantees uniqueness, while the surrogate key remains the stable identifier used in relationships.

Foreign keys

A **foreign key** is a column in one table that refers to the primary key of another. Each station belongs to a city, so `station.city_id` is a foreign key referencing `city.city_id`. The database enforces that every non-null value in `station.city_id` matches an existing city. You cannot create a station in city 99 if city 99 does not exist, and by default you cannot delete a city while stations still reference it.

This guarantee is called **referential integrity**, and it is one of the most valuable things a relational database does for you. Without it, "orphaned" rows accumulate — trips that reference deleted bikes, payments for members who no longer exist — and every report must defend against them. With it, a join from `trip` to `bike` can be trusted to find a bike for every trip.

> **TIP**
>
> When you explore an unfamiliar database, look at its foreign keys first. They are a map of how the tables relate and which joins are meaningful. In `psql`, the command `\d station` lists a table's columns, indexes, and foreign keys, including the tables that reference it.

Relationships Between Tables

Foreign keys express **relationships**. Three shapes cover nearly everything you will meet.

One-to-many

One city has many stations; each station belongs to exactly one city. This is a **one-to-many** relationship, and it is by far the most common shape. It is implemented by putting the foreign key on the "many" side: `station.city_id`. Spokeway is full of them: one member takes many trips, one bike has many maintenance tickets, one plan is chosen by many members.

One-to-one

Occasionally each row in one table matches at most one row in another, usually as a deliberate choice to split off wide or sensitive data. Spokeway has none in its core schema.

Many-to-many

Some relationships are many-to-many. A station is the start of many trips, and a bike is used on many trips; conversely each bike visits many stations over its life. A relational database represents a many-to-many relationship with a table in the middle that holds foreign keys to both sides. In Spokeway, `trip` plays that role for bikes and stations, while carrying its own facts as well (start time, end time, distance).

Here is the overall shape of the Spokeway schema. Each arrow points from the table holding a foreign key to the table it references.

```
station            --> city
bike               --> city (home_city_id)
member             --> city, plan
member             --> member (referred_by)
trip               --> member, bike, station (start), station (end)
payment            --> member, trip
maintenance_ticket --> bike, station
station_status     --> station
app_event              (stores member_id, but no foreign key)
```

Note that `member.referred_by` points back to `member` itself: a **self-referencing** relationship that records who referred whom, and that Chapter 9 follows with recursive queries. Sketching a map like this before writing a complex query is a habit that saves time, because it forces you to trace the path of

foreign keys between the tables your question involves.

The Relational Model in a Nutshell

The ideas above come from the **relational model**, a mathematical foundation proposed by Edgar F. Codd around 1970. You do not need the mathematics to write excellent SQL, but three of its principles explain behavior that otherwise looks arbitrary.

Everything is a relation

In the relational model, the result of every operation on relations is itself a relation. In SQL terms, every query produces a table-shaped result, and that result can be used anywhere a table can. This is why you can put a query inside another query (Chapter 8), give a query a name and refer to it later (Chapter 9), or save it as a view (Chapter 20). SQL is *compositional*: large queries are built from small ones, each of which you can run and check on its own.

Operations on sets

The core operations of the relational model map directly onto SQL clauses:

Relational operation	What it does	SQL expression
Selection (restriction)	Keep rows matching a condition	`WHERE`
Projection	Keep some columns	the `SELECT` list
Join	Combine rows from two relations	`JOIN ... ON`
Union	Rows in either relation	`UNION`
Difference	Rows in one but not the other	`EXCEPT`
Aggregation	Summarize groups of rows	`GROUP BY`

Note one historical quirk: what the relational model calls "selection" corresponds to SQL's `WHERE`, not to its `SELECT` keyword, which actually performs projection.

SQL is not a pure implementation

SQL departs from the pure relational model in a few ways that matter in practice. A mathematical relation is a *set*, so it cannot contain duplicate rows; an SQL table or query result is a *multiset* (or "bag") and can contain duplicates unless you prevent them with keys or remove them with `DISTINCT`. SQL also has `NULL`, a marker for missing or unknown values, which brings three-valued logic — true, false, and unknown — into every comparison. Chapter 4 is devoted largely to `NULL`, because it causes more subtle bugs than any other feature of the language.

Normalization, Briefly

Why does Spokeway store city names in a `city` table and refer to them by `city_id`, instead of writing "Lakeport" directly on every station and every member? The answer is **normalization**: organizing data so that each fact is stored in exactly one place.

Imagine a denormalized design where each trip row repeated the member's email, plan name, and plan price. If a member changed their email, you would have to update every one of their trips; miss one and the data contradicts itself. If the Monthly plan's price changed, historical trips would suddenly show the new price, which is probably wrong for billing. These are called **update anomalies**, and normalization prevents them.

Normalization has costs as well. Answering a question may require joining several tables, and analysts in particular often prefer wide, flattened tables for reporting. Real systems frequently keep a normalized transactional schema for the application and build denormalized reporting tables or views on top of it. Spokeway will do exactly that with an `analytics` schema in later chapters. Chapter 16 treats normalization and its trade-offs properly.

Transactional and Analytical Workloads

Our two audiences tend to work with two different kinds of database workload, and it helps to name them.

OLTP (online transaction processing) is the workload of applications: many small reads and writes, each touching a few rows, arriving concurrently. "Unlock bike 302 for member 1002" is a typical OLTP operation. It must be fast, it must be correct even when another rider tries to unlock the same bike at the same moment, and it must never be lost once confirmed.

OLAP (online analytical processing) is the workload of reporting and analysis: fewer queries, but each one reads large amounts of data and summarizes it. "Average trip duration by station and hour of day over the last year" is a typical OLAP query. It may scan millions of rows, and it rarely changes any.

Trait	OLTP (applications)	OLAP (analytics)
Typical query	Fetch or change a few rows	Scan and summarize many rows
Frequency	Very high, repeated	Lower, often ad hoc
Main concern	Latency, correctness	Throughput, expressiveness
Schema style	Normalized	Often denormalized
Typical storage	Row-oriented	Often column-oriented

PostgreSQL is primarily an OLTP database that also handles a great deal of analytical work well. Cloud data warehouses are built for OLAP, typically storing data by column and spreading work across many machines. The SQL looks remarkably similar on both.

> **NOTE**
>
> If you are an analyst working mainly in a cloud warehouse, everything in Parts I to III applies to you almost unchanged. Part V's material on PostgreSQL indexes is specific to row-oriented databases, but its core lessons — read the plan, understand what the engine scans, and avoid unnecessary work — carry over. Check your platform's documentation for its own tuning features.

How SQL Is Organized

SQL statements fall into a few families, each covered later in the book.

Family	Purpose	Main statements	Chapters
Query	Read data	`SELECT`	3–15
DML	Change data	`INSERT`, `UPDATE`, `DELETE`, `MERGE`	18
DDL	Define structure	`CREATE`, `ALTER`, `DROP`	17
DCL	Control access	`GRANT`, `REVOKE`	20, 26
TCL	Control transactions	`BEGIN`, `COMMIT`, `ROLLBACK`	19

A first look at a query

Here is a query you will be able to write comfortably by the end of Part II. It lists the three busiest start stations in one city during a single week.

busiest-stations-preview.sql

```
SELECT s.name              AS station_name,
       COUNT(*)            AS trips_started
FROM trip AS t
JOIN station AS s ON s.station_id = t.start_station_id
WHERE s.city_id = 1
  AND t.started_at >= TIMESTAMPTZ '2026-03-02 00:00:00+00'
  AND t.started_at <  TIMESTAMPTZ '2026-03-09 00:00:00+00'
GROUP BY s.name
ORDER BY trips_started DESC
LIMIT 3;
```

Even without knowing the details, you can probably read it: take trips, attach each trip's start station, keep trips in city 1 during that week, count trips per station name, sort by the count, and keep the top three. That readability is by design. SQL was intended to read somewhat like English, and well-formatted SQL remains one of the most readable ways to express a data question.

Logical order versus written order

SQL clauses are written in one order but *logically* evaluated in another. The written order is `SELECT`, `FROM`, `WHERE`, `GROUP BY`, `HAVING`, `ORDER BY`, `LIMIT`. The logical order is roughly:

1. `FROM` and `JOIN` — assemble the source rows.
2. `WHERE` — filter individual rows.
3. `GROUP BY` — form groups.
4. `HAVING` — filter groups.
5. `SELECT` — compute the output columns.
6. `DISTINCT` — remove duplicate output rows.
7. `ORDER BY` — sort.
8. `LIMIT` / `OFFSET` — keep a slice.

This logical order explains many rules that otherwise seem arbitrary. For example, you cannot refer to a column alias defined in `SELECT` from the `WHERE` clause, because `WHERE` is evaluated first; but you *can* refer to it in `ORDER BY`, which comes later. We will return to this list in Chapters 3 and 6. The database is free to execute the query physically in a different order — pushing filters down, reordering joins — as long as the result is the same as if it had followed the logical order.

What Goes Wrong in Practice

Most SQL problems in real teams are not exotic. They are the same handful of misunderstandings, repeated. Knowing them in advance is the fastest way to avoid them.

- **Misunderstanding the grain.** Someone joins `member` to `trip` and then sums a member-level value such as a sign-up bonus. Because each member now appears once per trip, the total is inflated. Always know what one row of your result represents. Chapter 7 covers this "fan-out" problem in depth.
- **Ignoring NULL.** A filter like "trips that did not end at station 3" silently drops trips whose `end_station_id` is `NULL`. Chapter 4 explains why.
- **Time zones.** Spokeway operates in several cities, and "trips on Monday" means different instants in each. Storing event times as `timestamptz` helps, but you still have to decide which time zone a report uses. Chapter 13 is devoted to this.
- **Trusting a query because it runs.** SQL rarely fails loudly on a logic error. A wrong join or filter returns a plausible number. Checking results against a small, known sample — like the one you will build in Chapter 2 — is a professional habit, not a beginner's crutch.

Practice

These exercises need no database yet; they build the mental model you will use from Chapter 3 onward.