Databases

Lecture Notes

Licence L3 — 2025–2026

Yaë Ulrich Gaba

"Data is the new oil."
— Clive Humby

March 25, 2026

Contents

Preface

Course Objectives

Databases are one of the fundamental pillars of modern computer science. From the early file management systems of the 1960s to the explosion of NoSQL databases and data lakes, the ability to store, organize, and query data reliably and efficiently remains a central challenge for every software application.

This course is designed for second-year undergraduate students (L2 / sophomore level) in computer science, mathematics-computer science, or data science. It assumes basic familiarity with programming (Python) and discrete mathematics (sets, relations, propositional logic).

Upon completion of this course, the student will be able to:

- Design a conceptual schema (Entity-Relationship model) and translate it into a normalized relational schema.
- Write complex SQL queries involving joins, aggregations, subqueries, and window functions.
- Understand and apply normal forms (1NF through BCNF) to eliminate update anomalies.
- Explain the ACID properties and concurrency control mechanisms.
- Choose and create appropriate indexes to optimize performance.
- Design a simple ETL pipeline using Python and SQL.
- Compare NoSQL models (document, key-value, column, graph) with the relational model.

Course Organization

The course is structured into eleven chapters, progressing from theoretical foundations to practical applications:

1. **Introduction to Relational Databases** — History, core concepts, DBMS architecture.
2. **Entity-Relationship Model** — Conceptual schema design, ER diagrams, mapping to relational schemas.

3. **Relational Algebra** — Formal operators: selection, projection, join, division.
4. **SQL — Basic Queries** — `SELECT`, `FROM`, `WHERE`, sorting and filtering.
5. **SQL — Joins and Subqueries** — Inner, outer, cross joins; correlated subqueries.
6. **SQL — Aggregation and Window Functions** — `GROUP BY`, `HAVING`, `ROW_NUMBER`, `RANK`.
7. **Normalization** — Functional dependencies, 1NF through BCNF, lossless decomposition.
8. **Transactions, Integrity, Constraints** — ACID properties, isolation levels, locking.
9. **Indexing and Query Optimization** — B-trees, hash indexes, query execution plans.
10. **Data Pipelines and ETL** — Extract, Transform, Load; Python/SQL integration.
11. **NoSQL Databases** — Document, key-value, column, graph; the CAP theorem.

Pedagogical Approach

Each chapter follows a consistent structure:

- **Theory** — Formal definitions, theorems, and propositions with proofs or justifications.
- **Detailed Examples** — Every concept is illustrated with one or more concrete examples, often using a "University" database as a running example.
- **Executable SQL Code** — Queries are presented with their expected output, testable on SQLite, PostgreSQL, or MySQL.
- **Python Integration** — `sqlite3` and `pandas` scripts demonstrate programmatic usage.
- **Exercises** — Exercises of increasing difficulty, from comprehension checks to open-ended problem solving.

Running Example Database

Throughout this course, we use a **University** database with the following tables:

University Database Schema

```sql
CREATE TABLE Students (
    student_id   INTEGER PRIMARY KEY,
    last_name    TEXT NOT NULL,
    first_name   TEXT NOT NULL,
    birth_date   DATE,
```

```
    major        TEXT
);

CREATE TABLE Courses (
    course_id    INTEGER PRIMARY KEY,
    title        TEXT NOT NULL,
    credits      INTEGER DEFAULT 3,
    prof_id      INTEGER REFERENCES Professors(prof_id)
);

CREATE TABLE Professors (
    prof_id      INTEGER PRIMARY KEY,
    last_name    TEXT NOT NULL,
    first_name   TEXT NOT NULL,
    department   TEXT
);

CREATE TABLE Enrollments (
    student_id   INTEGER REFERENCES Students(student_id),
    course_id    INTEGER REFERENCES Courses(course_id),
    grade        REAL,
    year         INTEGER,
    PRIMARY KEY (student_id, course_id)
);
```

This schema will be enriched throughout the chapters to illustrate advanced concepts (inheritance, ternary relationships, historization).

Sample Data

Inserting Sample Data

```
INSERT INTO Professors VALUES (1, 'Smith', 'Mary', 'Computer Science');
INSERT INTO Professors VALUES (2, 'Johnson', 'John', 'Mathematics');
INSERT INTO Professors VALUES (3, 'Williams', 'Sophie', 'Computer
↪  Science');

INSERT INTO Courses VALUES (101, 'Databases', 4, 1);
INSERT INTO Courses VALUES (102, 'Algorithms', 3, 2);
INSERT INTO Courses VALUES (103, 'Networks', 3, 3);
INSERT INTO Courses VALUES (104, 'Statistics', 3, 2);

INSERT INTO Students VALUES (1, 'Brown', 'Alice', '2004-03-15', 'CS');
INSERT INTO Students VALUES (2, 'Davis', 'Bob', '2003-11-22', 'CS');
INSERT INTO Students VALUES (3, 'Miller', 'Claire', '2004-07-08',
↪  'Math');
INSERT INTO Students VALUES (4, 'Wilson', 'David', '2003-01-30', 'CS');
INSERT INTO Students VALUES (5, 'Taylor', 'Emma', '2004-09-12', 'Math');
```

```sql
INSERT INTO Enrollments VALUES (1, 101, 88.5, 2025);
INSERT INTO Enrollments VALUES (1, 102, 76.0, 2025);
INSERT INTO Enrollments VALUES (2, 101, 72.0, 2025);
INSERT INTO Enrollments VALUES (2, 103, 91.0, 2025);
INSERT INTO Enrollments VALUES (3, 102, 95.5, 2025);
INSERT INTO Enrollments VALUES (3, 104, 82.0, 2025);
INSERT INTO Enrollments VALUES (4, 101, 55.0, 2025);
INSERT INTO Enrollments VALUES (5, 104, 97.0, 2025);
```

Recommended Tools

- **SQLite** — Lightweight, serverless, ideal for learning. Available via the `sqlite3` command line or the Python `sqlite3` module.
- **PostgreSQL** — Full-featured DBMS, SQL-standard compliant, recommended for advanced exercises (window functions, concurrent transactions).
- **DB Browser for SQLite** — GUI for visualizing and manipulating SQLite databases.
- **DBeaver** — Universal client supporting PostgreSQL, MySQL, SQLite, and others.
- **Python 3.x** — With modules `sqlite3` (standard library), `psycopg2` (PostgreSQL), `pandas` (data analysis).

Typographic Conventions

- SQL keywords are written in `UPPERCASE`: `SELECT`, `WHERE`, `JOIN`.
- Table and column names use `PascalCase` or `snake_case` depending on context.
- Technical terms appearing for the first time are in *italics*.
- Colored boxes indicate:
  - important **definitions** and **theorems**,
  - **best practices** to remember,
  - **anti-patterns** to avoid,
  - **warnings** about common pitfalls.

Prerequisites

- **Programming** — Ability to write simple Python programs (variables, loops, functions, file I/O).
- **Mathematics** — Basic set theory, relations, Cartesian products, propositional logic ($\wedge$, $\vee$, $\neg$, $\Rightarrow$).
- **Systems** — Familiarity with a command-line terminal.

Supplementary Resources

- A. Silberschatz, H. Korth, S. Sudarshan, *Database System Concepts*, McGraw-Hill, 7th edition.
- R. Elmasri, S. Navathe, *Fundamentals of Database Systems*, Pearson, 7th edition.
- R. Ramakrishnan, J. Gehrke, *Database Management Systems*, McGraw-Hill, 3rd edition.
- Official PostgreSQL Documentation: `https://www.postgresql.org/docs/`
- SQLite Documentation: `https://www.sqlite.org/docs.html`
- W3Schools SQL Tutorial: `https://www.w3schools.com/sql/`

Happy reading and happy learning!

Chapter 1

Introduction to Relational Databases

In 1970, Edgar F. Codd, a researcher at IBM, published a paper that would revolutionize computing: *A Relational Model of Data for Large Shared Data Banks*. His idea: organize data into *tables* connected by precise algebraic operations, radically separating how data is physically stored from how it is logically queried. IBM, in an irony of history, took years to implement its own researcher's ideas — it was a group at Berkeley, led by Michael Stonebraker, that created the first practical relational system. Today, relational databases store the bulk of the world's information.

1.1 What Is a Database?

Let us begin with the most fundamental question.

Definition 1.1 (Database)**.** A *database* is an organized collection of structured data, stored persistently and accessible by multiple users and applications in a controlled manner.

Unlike a simple flat file (CSV, text), a database provides:

- **Persistence** — data survives program termination.
- **Sharing** — multiple users access data concurrently.
- **Integrity** — constraints ensure data consistency.
- **Security** — access control mechanisms protect data.
- **Efficient querying** — a declarative language (SQL) lets users ask complex questions without specifying "how."

Example 1.2. Consider a university management system. Without a database, we might store students in a CSV file:

```
1,Brown,Alice,2004-03-15,CS
2,Davis,Bob,2003-11-22,CS
```

Problems: no type checking, no foreign keys, no safe concurrent access, no efficient complex queries.

1.2 A Brief History

1. **1960s** — File systems, hierarchical models (IBM's IMS) and network models (CODASYL). Data organized as trees or graphs with explicit navigation.
2. **1970** — Edgar F. Codd publishes *"A Relational Model of Data for Large Shared Data Banks,"* laying the foundations of the relational model. Data is organized in *tables* (relations) manipulated by set-based operators.
3. **1970s–80s** — Development of System R (IBM) and Ingres (Berkeley); birth of SQL (originally SEQUEL).
4. **1980s–90s** — Commercialization: Oracle, DB2, SQL Server, PostgreSQL. SQL standardization (SQL-86, SQL-92, SQL:1999).
5. **2000s** — Object-oriented databases, XML, data warehouses.
6. **2010s+** — NoSQL explosion (MongoDB, Cassandra, Neo4j), NewSQL, in-memory databases, data lakes.

Remark 1.3. Despite the rise of NoSQL, the relational model remains the most widely used in industry. SQL skills are among the most sought-after on the IT job market.

1.3 Database Management Systems (DBMS)

Definition 1.4 (DBMS)**.** A *Database Management System* (DBMS) is software that enables creating, managing, and querying databases. It ensures independence between application programs and the physical representation of data.

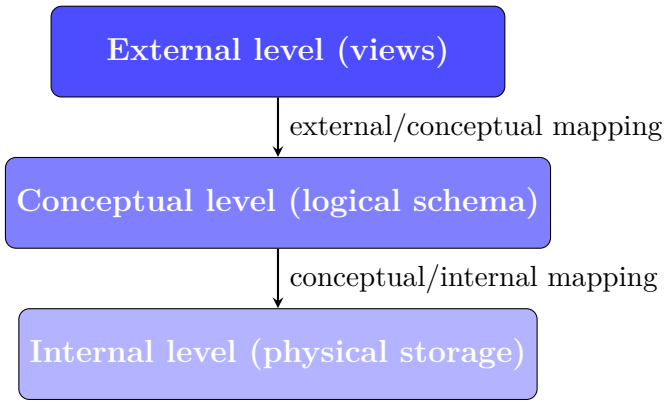
A DBMS provides the following functions (Codd's rules):

- **Data Definition** (DDL) — creating tables, constraints, indexes.
- **Data Manipulation** (DML) — inserting, updating, deleting, querying.
- **Data Control** (DCL) — access rights, roles.
- **Transaction Control** (TCL) — `BEGIN`, `COMMIT`, `ROLLBACK`.

1.3.1 Three-Level Architecture (ANSI/SPARC)

Definition 1.5 (ANSI/SPARC Architecture)**.** The ANSI/SPARC architecture (1975) defines three levels of abstraction:

1. **External level** (user views) — each user or application sees an adapted subset of the data.
2. **Conceptual level** (logical schema) — complete description of the data structure, independent of physical storage.
3. **Internal level** (physical schema) — on-disk organization: files, indexes, partitions.



Theorem 1.6 (Data Independence)**.** *The three-level architecture guarantees two forms of independence:*

- ***Logical independence****: the conceptual schema can be modified without affecting external views.*
- ***Physical independence****: the physical organization (adding indexes, changing file format) can be modified without affecting the conceptual schema.*

1.4 The Relational Model

Definition 1.7 (Relation)**.** Let $D_1, D_2, \ldots, D_n$ be domains (sets of possible values). A *relation* R with schema $R(A_1 : D_1, A_2 : D_2, \ldots, A_n : D_n)$ is a finite subset of the Cartesian product $D_1 \times D_2 \times \cdots \times D_n$. Each element of R is called a *tuple*.

Relational vs. Tabular Terminology

Relational Model	Table	SQL
Relation	Table	`TABLE`
Attribute	Column	`COLUMN`
Tuple	Row	`ROW`
Domain	Data type	`INTEGER`, `TEXT`, …
Degree	Number of columns	—
Cardinality	Number of rows	`COUNT(*)`

Definition 1.8 (Candidate Key and Primary Key)**.** Let R be a relation with schema $R(A_1, \ldots, A_n)$.

- A *superkey* of R is a subset $K \subseteq \{A_1, \ldots, A_n\}$ such that for any two distinct tuples $t_1, t_2 \in R$, we have $t_1[K] \neq t_2[K]$.
- A *candidate key* is a minimal superkey (no proper subset is itself a superkey).
- The *primary key* is the candidate key chosen by the designer to uniquely identify each tuple.

Example 1.9. In the `Students` table:

- {`student_id`} is a candidate key (and the chosen primary key).

- {last_name, first_name, birth_date} could be a candidate key if we assume uniqueness of this combination.

- {student_id, last_name} is a superkey but not a candidate key (not minimal).

Definition 1.10 (Foreign Key)**.** Given two relations R and S, a set of attributes FK of R is a *foreign key* referencing S if, for every tuple $t \in R$, $t[FK]$ is either NULL or equal to $s[PK]$ for some tuple $s \in S$, where PK is the primary key of S.

1.5 DBMS Languages

1.5.1 DDL — Data Definition Language

Creating a Table with Constraints

```sql
CREATE TABLE Students (
    student_id   INTEGER PRIMARY KEY,
    last_name    TEXT NOT NULL,
    first_name   TEXT NOT NULL,
    birth_date   DATE,
    major        TEXT CHECK (major IN ('CS', 'Math', 'Physics')),
    UNIQUE (last_name, first_name, birth_date)
);
```

1.5.2 DML — Data Manipulation Language

Basic CRUD Operations

```sql
-- Create (insert)
INSERT INTO Students (student_id, last_name, first_name, major)
VALUES (6, 'Garcia', 'Lucy', 'CS');

-- Read (query)
SELECT last_name, first_name FROM Students WHERE major = 'CS';

-- Update
UPDATE Students SET major = 'Math' WHERE student_id = 6;

-- Delete
DELETE FROM Students WHERE student_id = 6;
```

1.5.3 DCL — Data Control Language

Access Control (PostgreSQL)

```sql
-- Grant read access
GRANT SELECT ON Students TO web_user;

-- Grant all privileges
GRANT ALL PRIVILEGES ON Courses TO admin_user;

-- Revoke a privilege
REVOKE DELETE ON Students FROM web_user;
```

1.6 Major Relational DBMS

DBMS	License	Typical Use	Highlight
PostgreSQL	Open source	Enterprise, web	Highly SQL-compliant
MySQL/MariaDB	Open source	Web (LAMP)	Very widespread
SQLite	Public domain	Embedded, mobile	Serverless
Oracle DB	Commercial	Large enterprise	High performance
SQL Server	Commercial	Enterprise (.NET)	Microsoft integration

1.7 Getting Started with SQLite and Python

SQLite is built into Python via the `sqlite3` module:

Connecting and Creating Tables with Python

```python
import sqlite3

# Connect (creates the file if it does not exist)
conn = sqlite3.connect('university.db')
cur = conn.cursor()

# Create the table
cur.execute('''
    CREATE TABLE IF NOT EXISTS Students (
        student_id   INTEGER PRIMARY KEY,
        last_name    TEXT NOT NULL,
        first_name   TEXT NOT NULL,
        birth_date   DATE,
        major        TEXT
    )
''')

# Insert data
```

```
cur.execute("INSERT INTO Students VALUES (1, 'Brown', 'Alice',
↪  '2004-03-15', 'CS')")
cur.execute("INSERT INTO Students VALUES (2, 'Davis', 'Bob',
↪  '2003-11-22', 'CS')")

conn.commit()

# Query
cur.execute("SELECT * FROM Students")
for row in cur.fetchall():
    print(row)

conn.close()
```

Output

```
(1, 'Brown', 'Alice', '2004-03-15', 'CS')
(2, 'Davis', 'Bob', '2003-11-22', 'CS')
```

Use Parameterized Queries

Never build SQL queries by string concatenation: this exposes you to *SQL injection attacks*. Use parameter placeholders (`?`):

```
# GOOD: parameterized query
cur.execute("SELECT * FROM Students WHERE major = ?", ('CS',))

# BAD: string concatenation (SQL injection possible!)
# cur.execute("SELECT * FROM Students WHERE major = '" + major + "'")
```

1.8 SQL Data Types

Common SQL Data Types

Category	SQL Type	Description
Integer	`INTEGER`, `SMALLINT`, `BIGINT`	Whole numbers
Real	`REAL`, `FLOAT`, `DOUBLE`	Floating-point numbers
Decimal	`NUMERIC(p,s)`, `DECIMAL`	Exact precision
Text	`TEXT`, `VARCHAR(n)`, `CHAR(n)`	Character strings
Date/Time	`DATE`, `TIME`, `TIMESTAMP`	Temporal data
Boolean	`BOOLEAN`	True/false
Binary	`BLOB`	Binary data

NULL Is Not Zero

The value `NULL` represents the absence of a value. It differs from 0, the empty string `''`, or `FALSE`. Any comparison with `NULL` returns `UNKNOWN` (three-valued logic). Test for nullity using `IS NULL` or `IS NOT NULL`, never with `= NULL`.

1.9 Integrity Constraints

Definition 1.11 (Integrity Constraints)**.** Integrity constraints are rules that guarantee data consistency in the database. The main constraints are:

- `PRIMARY KEY` — uniquely identifies each row.
- `FOREIGN KEY` — references a primary key in another table.
- `NOT NULL` — forbids null values.
- `UNIQUE` — forbids duplicate values.
- `CHECK` — verifies a logical condition.
- `DEFAULT` — default value when none is provided.

Complete Constraint Example

```
CREATE TABLE Enrollments (
    student_id INTEGER NOT NULL,
    course_id  INTEGER NOT NULL,
    grade      REAL CHECK (grade >= 0 AND grade <= 100),
    year       INTEGER DEFAULT 2025,
    PRIMARY KEY (student_id, course_id),
    FOREIGN KEY (student_id) REFERENCES Students(student_id)
        ON DELETE CASCADE,
    FOREIGN KEY (course_id) REFERENCES Courses(course_id)
        ON UPDATE CASCADE
);
```

1.10 Data Models: An Overview

Besides the relational model, other data models exist:

Model	Structure	Example DBMS
Relational	Tables (rows/columns)	PostgreSQL, MySQL
Document	JSON/BSON documents	MongoDB, CouchDB
Key-Value	Key → value pairs	Redis, DynamoDB
Column	Column families	Cassandra, HBase
Graph	Nodes and edges	Neo4j, ArangoDB

Chapter 11 will explore these alternatives in depth.

Exercises

Exercise 1.1. Give three advantages of a DBMS over a flat file system (CSV). Illustrate each advantage with a concrete example related to university management.

Exercise 1.2. For each of the following tables, identify the candidate key(s):

1. `Books(isbn, title, author, publication_year)`
2. `Employees(badge_id, last_name, first_name, email, department)`
3. `Flights(airline, flight_number, departure_date, origin, destination)`

Exercise 1.3. Write SQL code to create a library database with the tables `Books`, `Authors`, `Loans`, and `Members`. Define primary keys, foreign keys, and at least two `CHECK` constraints.

Exercise 1.4. Write a Python script using `sqlite3` that:

1. Creates the database `library.db`.
2. Creates the tables defined in the previous exercise.
3. Inserts at least 5 books and 3 members.
4. Displays all books published after 2020.

Exercise 1.5. Explain the difference between logical data independence and physical data independence in the ANSI/SPARC architecture. Give a concrete example for each type.

Chapter 2

The Entity-Relationship Model

In 1976, Peter Chen, a young computer scientist at MIT, published a paper that would become one of the most cited in the history of database research: "The Entity-Relationship Model—Toward a Unified View of Data." At the time, database design was an ad hoc affair. Programmers jumped straight into tables and columns, only to discover months later that their schemas could not accommodate new requirements. Chen's insight was to introduce an intermediate, conceptual layer—a diagram that captures the *meaning* of data before committing to any particular implementation.

The Entity-Relationship (ER) model is, at its core, a language for thinking about the world in terms of *things* (entities), their *properties* (attributes), and the *connections* between them (relationships). It is deliberately independent of any database management system: an ER diagram describes what the data *is*, not how it is stored. This separation of concerns—conceptual design first, logical design second—remains the gold standard of database engineering. In this chapter, we learn to speak this language fluently.

2.1 Introduction to Database Design

Database design follows three main steps:

1. **Requirements analysis** — gathering user requirements.
2. **Conceptual design** — modeling independent of any DBMS (Entity-Relationship model).
3. **Logical design** — translation into a relational schema (SQL tables).
4. **Physical design** — choosing indexes, partitioning, etc.

The Entity-Relationship (ER) model, introduced by Peter Chen in 1976, is the standard tool for conceptual design.

2.2 Core Concepts

Definition 2.1 (Entity)**.** An *entity* is a real-world object, distinguishable from other objects, about which we wish to store information. An *entity type* groups entities sharing the same attributes.

CHAPTER 2. THE ENTITY-RELATIONSHIP MODEL

Definition 2.2 (Attribute)**.** An *attribute* is a property that describes an entity or a relationship. Each attribute has a *domain* (set of possible values). We distinguish:

- **Simple** vs. **composite** (e.g., address = street + city + zip).
- **Single-valued** vs. **multi-valued** (e.g., phone numbers).
- **Stored** vs. **derived** (e.g., age computed from birth date).

Definition 2.3 (Entity Key)**.** An attribute (or set of attributes) whose value uniquely identifies each instance of an entity type. It is underlined in the ER diagram.

Definition 2.4 (Relationship)**.** A *relationship* represents a link between two or more entity types. It may have its own attributes.

2.3 Cardinalities

Definition 2.5 (Cardinality)**.** The *cardinality* of an entity type's participation in a relationship expresses the minimum and maximum number of relationship instances in which an entity instance can participate. Notation: (min, max).

The most common cardinalities are:

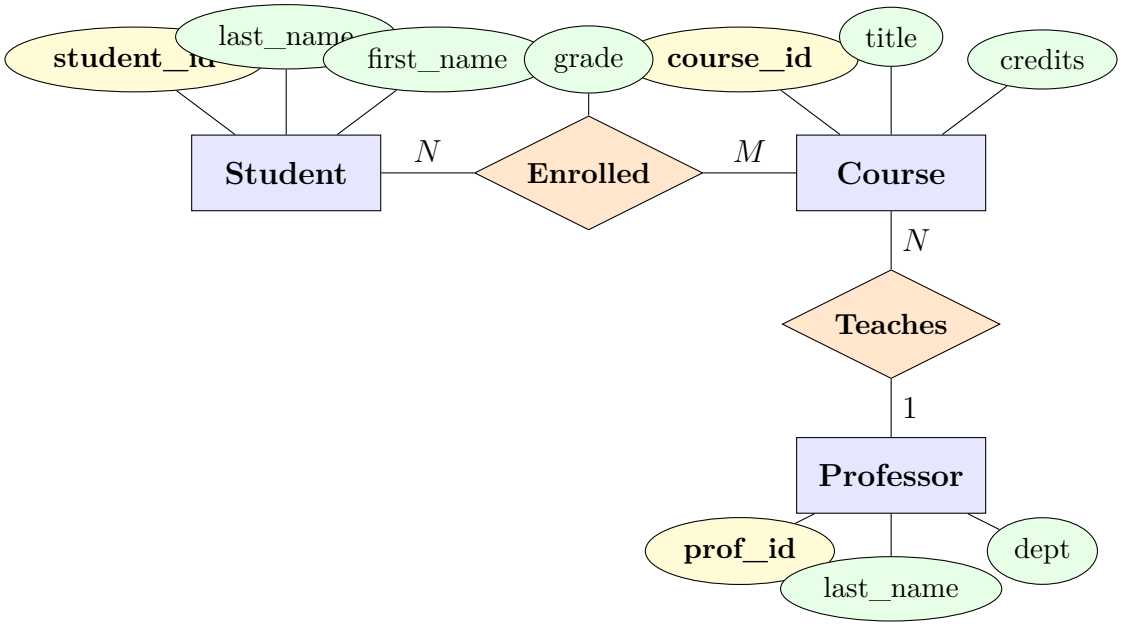
Cardinality Types

Notation	Meaning	Example
$1:1$ (one-to-one)	Each entity links to at most one	Person — Passport
$1:N$ (one-to-many)	One entity links to many	Professor — Course
$N:M$ (many-to-many)	Many to many	Student — Course

Example 2.6. A professor teaches *several* courses, but each course is taught by *one* professor: this is a $1:N$ relationship.

A student can enroll in *several* courses, and each course has *several* students: this is an $N:M$ relationship.

2.4 ER Diagrams with TikZ



2.5 Weak Entities

Definition 2.7 (Weak Entity)**.** A *weak entity* is an entity type that does not have a sufficient key of its own to uniquely identify its instances. Its identification depends on an *owner* (strong) entity type via an *identifying relationship*.

Example 2.8. Consider the rooms in a building. A room is identified by its number *within* a given building. The `Room` entity type is weak, identified by the combination of `Building.building_id` and `Room.room_number`.

2.6 Specialization and Generalization

Definition 2.9 (Specialization / Generalization)**.** *Specialization* consists of defining subtypes of a general entity type ("is-a" relationship). *Generalization* is the reverse process: grouping entity types that share common attributes into a supertype.

Example 2.10. The type `Person` can be specialized into `Student` and `Professor`. Common attributes (name) go in `Person`; specific attributes (major for Student, department for Professor) go in the subtypes.

Specialization constraints:

- **Disjoint** (d) vs. **overlapping** (o): can an entity belong to multiple subtypes?
- **Total** (t) vs. **partial** (p): must every supertype entity belong to at least one subtype?

2.7 Mapping ER to Relational Schema

Theorem 2.11 (ER-to-Relational Translation Rules)**.** *The mapping rules are as follows:*

1. ***Strong entity type*** $\rightarrow$ *a table whose primary key is the entity key.*
2. ***Weak entity type*** $\rightarrow$ *a table whose primary key combines the partial key and the owner's key (foreign key).*
3. $1:N$ ***relationship*** $\rightarrow$ *add a foreign key in the table on the N side.*
4. $N:M$ ***relationship*** $\rightarrow$ *create an association table whose primary key is the pair of foreign keys.*
5. $1:1$ ***relationship*** $\rightarrow$ *foreign key in one of the two tables (preferably the one with total participation).*
6. ***Multi-valued attribute*** $\rightarrow$ *separate table with a foreign key.*
7. ***Specialization*** $\rightarrow$ *several strategies possible (separate tables, single table with discriminator, etc.).*

Example 2.12. Applying the rules to the previous diagram:

Translating the ER Diagram to SQL

```sql
-- Rule 1: strong entity -> table
CREATE TABLE Students (
    student_id  INTEGER PRIMARY KEY,
    last_name   TEXT NOT NULL,
    first_name  TEXT NOT NULL
);

CREATE TABLE Courses (
    course_id   INTEGER PRIMARY KEY,
    title       TEXT NOT NULL,
    credits     INTEGER DEFAULT 3,
    prof_id     INTEGER REFERENCES Professors(prof_id)  -- Rule 3: 1:N
);

CREATE TABLE Professors (
    prof_id     INTEGER PRIMARY KEY,
    last_name   TEXT NOT NULL,
    department  TEXT
);

-- Rule 4: N:M relationship -> association table
CREATE TABLE Enrollments (
    student_id  INTEGER REFERENCES Students(student_id),
    course_id   INTEGER REFERENCES Courses(course_id),
    grade       REAL,        -- relationship attribute
    PRIMARY KEY (student_id, course_id)
);
```

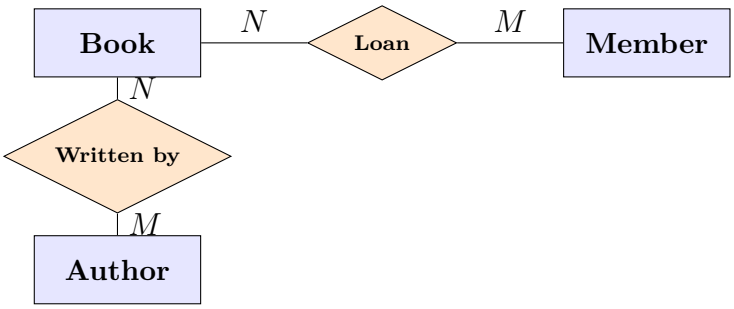
2.8 Complete Example: Library System

2.8.1 Requirements Analysis

A library needs to manage:

- **Books** (ISBN, title, publication year).
- **Authors** (name, nationality). A book can have multiple authors.
- **Members** (member number, name, address).
- **Loans** (loan date, due date, actual return date).

2.8.2 ER Diagram



2.8.3 SQL Translation

Library Relational Schema

```sql
CREATE TABLE Authors (
    author_id    INTEGER PRIMARY KEY,
    name         TEXT NOT NULL,
    nationality  TEXT
);

CREATE TABLE Books (
    isbn         TEXT PRIMARY KEY,
    title        TEXT NOT NULL,
    pub_year     INTEGER
);

CREATE TABLE Written_By (
    isbn         TEXT REFERENCES Books(isbn),
    author_id    INTEGER REFERENCES Authors(author_id),
    PRIMARY KEY (isbn, author_id)
);

CREATE TABLE Members (
    member_id    INTEGER PRIMARY KEY,
    name         TEXT NOT NULL,
    address      TEXT
```

```
);

CREATE TABLE Loans (
    loan_id      INTEGER PRIMARY KEY,
    isbn         TEXT REFERENCES Books(isbn),
    member_id    INTEGER REFERENCES Members(member_id),
    loan_date    DATE NOT NULL,
    due_date     DATE NOT NULL,
    return_date  DATE
);
```

2.9 Design Best Practices

Golden Rules of ER Design

1. Every entity type must have a well-defined key.
2. Avoid multi-valued attributes: convert them to separate entities.
3. Prefer binary relationships over ternary ones (easier to understand and translate).
4. Do not confuse entity and attribute: if a concept has its own attributes or participates in relationships, it is an entity.
5. Document every design decision (cardinalities, constraints).

Anti-pattern: The "Mega-Table"

Putting all information into a single giant table (students, courses, grades, professors in one table) causes:

- Massive data redundancy.
- Insertion, update, and deletion anomalies.
- Maintenance and evolution difficulties.

Normalization (Chapter 7) will formalize these problems.

2.10 Python Integration: Schema Generation

Automatic Schema Generation with Python

```
import sqlite3

schema = {
```

```
    'Authors': {
        'author_id': 'INTEGER PRIMARY KEY',
        'name': 'TEXT NOT NULL',
        'nationality': 'TEXT'
    },
    'Books': {
        'isbn': 'TEXT PRIMARY KEY',
        'title': 'TEXT NOT NULL',
        'pub_year': 'INTEGER'
    }
}

conn = sqlite3.connect('library.db')
cur = conn.cursor()

for table, columns in schema.items():
    cols = ', '.join(f'{col} {typ}' for col, typ in columns.items())
    sql = f'CREATE TABLE IF NOT EXISTS {table} ({cols})'
    print(f'Executing: {sql}')
    cur.execute(sql)

conn.commit()
conn.close()
```

Exercises

Exercise 2.1. Design an ER diagram for a hotel reservation system with the entities `Guest`, `Room`, `Hotel`, and the relationship `Reservation`. Specify cardinalities and attributes for each entity and relationship.

Exercise 2.2. Translate the ER diagram from the previous exercise into a relational SQL schema. Identify all primary and foreign keys.

Exercise 2.3. A hospital manages `Patients`, `Doctors`, `Departments`, and `Appointments`. A patient can see multiple doctors; a doctor belongs to exactly one department.

1. Draw the ER diagram.
2. Identify a potential weak entity.
3. Translate to SQL.

Exercise 2.4. Which specialization translation strategy would you choose to model a system where `Vehicle` is specialized into `Car` and `Truck`, with disjoint and total specialization? Justify your choice and provide the corresponding SQL code.

Exercise 2.5. Identify and correct the design errors in the following schema:

```
CREATE TABLE Orders (
    order_id INTEGER PRIMARY KEY,
```

```
    customer_name TEXT,
    customer_address TEXT,
    customer_phone TEXT,
    product TEXT,
    price REAL,
    quantity INTEGER,
    supplier_name TEXT,
    supplier_address TEXT
);
```

Propose a correct ER diagram and the corresponding SQL schema.

Chapter 3

Relational Algebra

3.1 Introduction

When you write `SELECT name FROM students WHERE grade > 15`, you are thinking in SQL. But behind this syntax lies a mathematical language of remarkable precision: *relational algebra*, invented by Edgar F. Codd in his landmark 1970 paper. Codd had the brilliant insight to view tables as *relations* in the mathematical sense (subsets of Cartesian products) and queries as *operations* on these relations: selection, projection, join, union, difference. Relational algebra is to SQL what formal logic is to natural language: a rigorous foundation that guarantees every query has a precise, unambiguous semantics.

Understanding relational algebra is understanding *why* SQL works—and how the query optimiser transforms your instructions into an efficient execution plan.

Definition 3.1 (Relational Algebra)**.** *Relational algebra* is a set of operators that take one or two relations as input and produce a relation as output. It is a *procedural* language: one specifies *how* to obtain the result (sequence of operations).

The operators fall into two categories:

- **Set operators**: union ($\cup$), intersection ($\cap$), difference ($-$), Cartesian product ($\times$).
- **Relation-specific operators**: selection (σ), projection (π), join ($\bowtie$), rename (ρ), division ($\div$).

3.2 Selection (σ)

Definition 3.2 (Selection)**.** The *selection* $\sigma_\theta(R)$ returns the tuples of R that satisfy the condition θ. The condition θ is a Boolean expression on the attributes of R.

$$\sigma_\theta(R) = \{t \in R \mid \theta(t) = \text{true}\}$$

Properties of Selection

$$\sigma_{\theta_1 \wedge \theta_2}(R) = \sigma_{\theta_1}(\sigma_{\theta_2}(R)) \qquad \text{(cascade)}$$
$$\sigma_{\theta_1}(\sigma_{\theta_2}(R)) = \sigma_{\theta_2}(\sigma_{\theta_1}(R)) \qquad \text{(commutativity)}$$
$$|\sigma_\theta(R)| \leq |R| \qquad \text{(cardinality)}$$

Example 3.3. Select CS students:

$$\sigma_{\text{major='CS'}}(\text{Students})$$

SQL equivalent:

SQL Equivalent of Selection

```
SELECT * FROM Students WHERE major = 'CS';
```

Output

```
student_id | last_name | first_name | birth_date | major
-----------+-----------+------------+------------+------
1          | Brown     | Alice      | 2004-03-15 | CS
2          | Davis     | Bob        | 2003-11-22 | CS
4          | Wilson    | David      | 2003-01-30 | CS
```

3.3 Projection (π)

Definition 3.4 (Projection)**.** The *projection* $\pi_{A_1,A_2,\ldots,A_k}(R)$ returns the relation obtained by keeping only attributes $A_1, \ldots, A_k$ of R and eliminating duplicates.

$$\pi_{A_1,\ldots,A_k}(R) = \{t[A_1, \ldots, A_k] \mid t \in R\}$$

Example 3.5. Project the names of students:

$$\pi_{\text{last\_name,first\_name}}(\text{Students})$$

SQL Equivalent of Projection

```
SELECT DISTINCT last_name, first_name FROM Students;
```

Projection and Duplicates

In relational algebra, projection automatically eliminates duplicates (since a relation is a set). In SQL, you must explicitly use `DISTINCT`, because SQL works with *multisets* (bags).

3.4 Set Operators

Definition 3.6 (Union, Intersection, Difference)**.** Let R and S be two relations with the same schema (*union-compatible*):

$$R \cup S = \{t \mid t \in R \vee t \in S\} \quad \text{(union)}$$
$$R \cap S = \{t \mid t \in R \wedge t \in S\} \quad \text{(intersection)}$$
$$R - S = \{t \mid t \in R \wedge t \notin S\} \quad \text{(difference)}$$

Theorem 3.7 (Set Properties).

- $\cup$ *and* $\cap$ *are commutative and associative.*
- $-$ *is neither commutative nor associative.*
- $R \cap S = R - (R - S)$.

Set Operators in SQL

```sql
-- Union (eliminates duplicates)
SELECT last_name FROM Students WHERE major = 'CS'
UNION
SELECT last_name FROM Students WHERE major = 'Math';

-- Intersection
SELECT student_id FROM Enrollments WHERE course_id = 101
INTERSECT
SELECT student_id FROM Enrollments WHERE course_id = 102;

-- Difference
SELECT student_id FROM Enrollments WHERE course_id = 101
EXCEPT
SELECT student_id FROM Enrollments WHERE course_id = 103;
```

3.5 Cartesian Product (×)

Definition 3.8 (Cartesian Product). The *Cartesian product* $R \times S$ produces all possible combinations of tuples from R with tuples from S.

$$R \times S = \{(t_r, t_s) \mid t_r \in R \wedge t_s \in S\}$$

If $|R| = n$ and $|S| = m$, then $|R \times S| = n \times m$.

Cost of the Cartesian Product

The Cartesian product is very expensive: it multiplies the cardinalities. In practice, it is rarely used alone; it is combined with a selection to form a *join*.

3.6 Join (⋈)

Definition 3.9 (Natural Join). The *natural join* $R \bowtie S$ combines tuples from R and S that have equal values on their common attributes.

$$R \bowtie S = \sigma_{\text{common attributes equal}}(R \times S)$$

with elimination of duplicate columns.

Definition 3.10 (Theta-Join). The *theta-join* $R \bowtie_\theta S$ is defined as:

$$R \bowtie_\theta S = \sigma_\theta(R \times S)$$

where θ is any condition on the attributes of R and S. Special case: an *equi-join* uses only equality conditions.

Example 3.11. Find the names of students and the titles of courses they are enrolled in:

$$\pi_{\text{last\_name,title}}(\text{Students} \bowtie \text{Enrollments} \bowtie \text{Courses})$$

Join in SQL

```sql
SELECT s.last_name, c.title
FROM Students s
JOIN Enrollments e ON s.student_id = e.student_id
JOIN Courses c ON e.course_id = c.course_id;
```

Output

```
last_name | title
----------+-----------
Brown     | Databases
Brown     | Algorithms
Davis     | Databases
Davis     | Networks
Miller    | Algorithms
Miller    | Statistics
Wilson    | Databases
Taylor    | Statistics
```

Proposition 3.12 (Properties of the Join)**.**

- The natural join is commutative: $R \bowtie S = S \bowtie R$.
- The natural join is associative: $(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$.
- If R and S have no common attributes, $R \bowtie S = R \times S$.

3.7 Rename (ρ)

Definition 3.13 (Rename)**.** The *rename* operator $\rho_{B/A}(R)$ renames attribute A to B in relation R. One can also rename the relation itself: $\rho_S(R)$.

Renaming is useful for:

- Making two relations union-compatible.
- Performing a self-join (joining a relation with itself).

Example 3.14. Find pairs of students enrolled in the same course:

$$\pi_{e_1.\text{student\_id},e_2.\text{student\_id}}(\sigma_{e_1.\text{student\_id}<e_2.\text{student\_id}}(\rho_{e_1}(\text{Enrollments}) \bowtie_{\text{course\_id}} \rho_{e_2}(\text{Enrollments})))$$

3.8 Division (÷)

Definition 3.15 (Division)**.** Let $R(A, B)$ and $S(B)$ be two relations. The *division* $R \div S$ returns the values of A in R that are associated with *all* values of B in S.

$$R \div S = \{t[A] \mid t \in R \wedge \forall s \in S, (t[A], s[B]) \in R\}$$

Theorem 3.16 (Division Expression)**.** *Division can be expressed using other operators:*

$$R \div S = \pi_A(R) - \pi_A\big((\pi_A(R) \times S) - R\big)$$

Example 3.17. Find students enrolled in *all* courses taught by professor 2:
Let $S = \pi_{\text{course\_id}}(\sigma_{\text{prof\_id}=2}(\text{Courses}))$ (courses taught by professor 2).
$\pi_{\text{student\_id,course\_id}}(\text{Enrollments}) \div S$
gives the students enrolled in all those courses.

Division in SQL (NOT EXISTS)

```sql
SELECT DISTINCT e.student_id
FROM Enrollments e
WHERE NOT EXISTS (
    SELECT c.course_id
    FROM Courses c
    WHERE c.prof_id = 2
    AND NOT EXISTS (
        SELECT 1
        FROM Enrollments e2
        WHERE e2.student_id = e.student_id
        AND e2.course_id = c.course_id
    )
);
```

3.9 Query Trees and Optimization

A relational algebra expression can be represented as a tree: leaves are base relations, internal nodes are operators.

Theorem 3.18 (Heuristic Optimization Rules)**.**

1. ***Push selections down****: apply selections as early as possible to reduce the size of intermediate relations.*
2. ***Push projections down****: project as early as possible to reduce the number of attributes.*
3. ***Combine selection + Cartesian product into join****:* $\sigma_\theta(R \times S) \to R \bowtie_\theta S$.
4. ***Reorder joins****: start with joins that produce the smallest intermediate results.*

Example 3.19. Consider the query: "names of CS students enrolled in course 101."
Naive expression:

$$\pi_{\text{last\_name}}(\sigma_{\text{major='CS'} \wedge \text{course\_id}=101}(\text{Students} \times \text{Enrollments}))$$

Optimized expression:

$$\pi_{\text{last\_name}}(\sigma_{\text{major='CS'}}(\text{Students}) \bowtie \sigma_{\text{course\_id}=101}(\text{Enrollments}))$$

The optimized version avoids the full Cartesian product and filters early.

3.10 Algebra–SQL Correspondence Summary

Relational Algebra ↔ SQL Mapping

Algebra	SQL
$\sigma_\theta(R)$	`SELECT * FROM R WHERE` θ
$\pi_{A_1,\ldots,A_k}(R)$	`SELECT DISTINCT` A_1, …, A_k `FROM R`
$R \cup S$	`R UNION S`
$R \cap S$	`R INTERSECT S`
$R - S$	`R EXCEPT S`
$R \times S$	`SELECT * FROM R, S`
$R \bowtie S$	`SELECT * FROM R NATURAL JOIN S`
$R \bowtie_\theta S$	`SELECT * FROM R JOIN S ON` θ
$R \div S$	Double `NOT EXISTS`
$\rho_{B/A}(R)$	`SELECT A AS B FROM R`

3.11 Python Integration

Simulating Relational Algebra with pandas

```python
import pandas as pd

# Relations as DataFrames
students = pd.DataFrame({
    'student_id': [1, 2, 3, 4, 5],
    'last_name': ['Brown', 'Davis', 'Miller', 'Wilson', 'Taylor'],
    'major': ['CS', 'CS', 'Math', 'CS', 'Math']
})
enrollments = pd.DataFrame({
    'student_id': [1, 1, 2, 2, 3, 3, 4, 5],
    'course_id': [101, 102, 101, 103, 102, 104, 101, 104],
    'grade': [88.5, 76.0, 72.0, 91.0, 95.5, 82.0, 55.0, 97.0]
})

# Selection: sigma_{major='CS'}(Students)
cs = students[students['major'] == 'CS']
print("Selection:\n", cs)

# Projection: pi_{last_name}(Students)
names = students[['last_name']].drop_duplicates()
print("\nProjection:\n", names)
```

```
# Natural join
result = pd.merge(students, enrollments, on='student_id')
print("\nJoin:\n", result[['last_name', 'course_id', 'grade']])
```

Exercises

Exercise 3.1. Express the following queries in relational algebra, then translate them to SQL:

1. Names of professors in the "Computer Science" department.
2. Titles of courses worth at least 4 credits.
3. Names of students enrolled in the "Databases" course.

Exercise 3.2. Let $R(A, B, C)$ and $S(C, D, E)$. Simplify the expression:

$$\pi_{A,D}(\sigma_{B>10 \wedge D='x'}(R \bowtie S))$$

by pushing selections and projections down.

Exercise 3.3. Express the following division in SQL: "find students enrolled in all courses from the Computer Science department."

Exercise 3.4. Draw the algebraic tree corresponding to the expression:

$$\pi_{\text{last\_name}}(\sigma_{\text{grade}>85}(\text{Students} \bowtie \text{Enrollments} \bowtie \text{Courses}))$$

Then propose an optimized tree by applying heuristic rules.

Exercise 3.5. Prove that $R \cap S = R - (R - S)$ using the set-theoretic definitions of the operators.

Chapter 4

SQL — Basic Queries

In 1974, Donald Chamberlin and Raymond Boyce, researchers at IBM, created SEQUEL (*Structured English Query Language*), later renamed SQL for legal reasons. Their ambition: to allow non-programmers to query databases by writing sentences close to English. The idea was revolutionary: instead of saying *how* to traverse the data (loops, indexes, pointers), you simply say *what* you want. The system finds the most efficient path. Fifty years later, SQL remains the universal language of relational databases, used by millions of developers every day.

4.1 Introduction to SQL

Definition 4.1 (SQL)**.** *SQL* (*Structured Query Language*) is the standard language for querying and manipulating relational databases. Unlike relational algebra, SQL is a *declarative* language: you specify *what* you want, not *how* to get it.

SQL has been standardized by ISO/ANSI. Major versions include: SQL-86, SQL-92, SQL:1999, SQL:2003, SQL:2011, SQL:2016, SQL:2023.

4.2 Structure of a SELECT Query

General SELECT Syntax

```
SELECT [DISTINCT] columns
FROM   tables
[WHERE  condition]
[ORDER BY column [ASC|DESC]]
[LIMIT  n [OFFSET m]];
```

The logical execution order differs from the writing order:

1. `FROM` — determine source tables.
2. `WHERE` — filter rows.
3. `SELECT` — project columns.
4. `DISTINCT` — eliminate duplicates.

5. `ORDER BY` — sort the result.

6. `LIMIT / OFFSET` — limit the number of rows.

4.3 SELECT and FROM

Selecting All Columns

```
SELECT * FROM Students;
```

Output

```
student_id | last_name | first_name | birth_date | major
-----------+-----------+------------+------------+------
1          | Brown     | Alice      | 2004-03-15 | CS
2          | Davis     | Bob        | 2003-11-22 | CS
3          | Miller    | Claire     | 2004-07-08 | Math
4          | Wilson    | David      | 2003-01-30 | CS
5          | Taylor    | Emma       | 2004-09-12 | Math
```

Selecting Specific Columns

```
SELECT last_name, first_name, major FROM Students;
```

Avoid SELECT *

In production, avoid `SELECT *`:

- It transfers unnecessary data.
- Results depend on the schema: adding a column may break code.
- Use `SELECT *` only for exploration and debugging.

4.4 Column and Table Aliases

Using Aliases

```
SELECT s.last_name AS student_name,
       s.first_name AS student_first,
       s.major AS department
FROM Students AS s;
```

The `AS` keyword is optional in most DBMS: `SELECT last_name student_name FROM Students s`.

4.5 WHERE Clause — Filtering

Definition 4.2 (WHERE Clause)**.** The `WHERE` clause filters rows based on a Boolean condition. Only rows for which the condition evaluates to `TRUE` are kept (`FALSE` and `UNKNOWN` are rejected).

4.5.1 Comparison Operators

SQL Comparison Operators

Operator	Meaning
`=`	Equal
`<>` or `!=`	Not equal
`<`, `>`	Less than, greater than
`<=`, `>=`	Less/greater than or equal
`BETWEEN a AND b`	Between a and b (inclusive)
`IN (v1, v2, ...)`	Member of a list
`LIKE 'pattern'`	Pattern matching
`IS NULL` / `IS NOT NULL`	Null test

Filtering Examples

```sql
-- CS students
SELECT * FROM Students WHERE major = 'CS';

-- Grades between 70 and 90
SELECT * FROM Enrollments WHERE grade BETWEEN 70 AND 90;

-- Names starting with 'B'
SELECT * FROM Students WHERE last_name LIKE 'B%';

-- Students in CS or Math
SELECT * FROM Students WHERE major IN ('CS', 'Math');

-- Enrollments with no grade
SELECT * FROM Enrollments WHERE grade IS NULL;
```

4.5.2 Logical Operators

Combining Conditions

```sql
-- AND: both conditions must be true
SELECT * FROM Students
WHERE major = 'CS' AND birth_date > '2004-01-01';

-- OR: at least one condition true
SELECT * FROM Enrollments
```

```sql
WHERE grade > 90 OR grade < 50;

-- NOT: negation
SELECT * FROM Students
WHERE NOT major = 'Math';
```

Three-Valued Logic and NULL

SQL uses *three-valued logic*: `TRUE`, `FALSE`, `UNKNOWN`.

AND	T	F	U
T	T	F	U
F	F	F	F
U	U	F	U

OR	T	F	U
T	T	T	T
F	T	F	U
U	T	U	U

`NOT UNKNOWN = UNKNOWN`. Consequence: `grade = NULL` returns `UNKNOWN`, not `TRUE`. Use `IS NULL`.

4.5.3 LIKE Patterns

LIKE Wildcards

Wildcard	Meaning
%	Zero or more arbitrary characters
_	Exactly one arbitrary character

LIKE Examples

```sql
-- Names starting with 'M'
SELECT * FROM Students WHERE last_name LIKE 'M%';

-- First names with exactly 5 characters
SELECT * FROM Students WHERE first_name LIKE '_____';

-- Names containing 'il'
SELECT * FROM Students WHERE last_name LIKE '%il%';
```

4.6 ORDER BY — Sorting

Sorting Results

```sql
-- Ascending sort (default)
SELECT last_name, first_name FROM Students ORDER BY last_name ASC;

-- Descending sort
```

```sql
SELECT * FROM Enrollments ORDER BY grade DESC;

-- Multi-column sort
SELECT * FROM Students ORDER BY major ASC, last_name ASC;
```

Output

```
-- Sorted by grade descending:
student_id | course_id | grade | year
-----------+-----------+-------+-----
5          | 104       | 97.0  | 2025
3          | 102       | 95.5  | 2025
2          | 103       | 91.0  | 2025
1          | 101       | 88.5  | 2025
3          | 104       | 82.0  | 2025
1          | 102       | 76.0  | 2025
2          | 101       | 72.0  | 2025
4          | 101       | 55.0  | 2025
```

4.7 DISTINCT — Eliminating Duplicates

Using DISTINCT

```sql
-- Without DISTINCT: duplicates possible
SELECT major FROM Students;
-- CS, CS, Math, CS, Math

-- With DISTINCT: unique values
SELECT DISTINCT major FROM Students;
-- CS, Math
```

4.8 LIMIT and OFFSET — Pagination

Limiting Results

```sql
-- Top 3 grades
SELECT s.last_name, e.grade
FROM Students s
JOIN Enrollments e ON s.student_id = e.student_id
ORDER BY e.grade DESC
LIMIT 3;

-- Pagination: page 2 (rows 4 to 6)
SELECT * FROM Students ORDER BY student_id LIMIT 3 OFFSET 3;
```

Output

```
-- Top 3 grades:
last_name | grade
----------+------
Taylor    | 97.0
Miller    | 95.5
Davis     | 91.0
```

LIMIT Is Not Standard SQL

`LIMIT` is supported by PostgreSQL, MySQL, SQLite. The SQL:2008 standard uses `FETCH FIRST n ROWS ONLY`:

```
SELECT * FROM Students
ORDER BY last_name
FETCH FIRST 3 ROWS ONLY;  -- Standard SQL
```

4.9 Expressions and Scalar Functions

Computed Expressions

```
-- Arithmetic expression
SELECT last_name, grade, ROUND(grade / 100.0 * 20, 1) AS grade_20
FROM Enrollments e
JOIN Students s ON e.student_id = s.student_id;

-- String concatenation
SELECT last_name || ' ' || first_name AS full_name
FROM Students;

-- Conditional expression CASE
SELECT last_name, grade,
    CASE
        WHEN grade >= 90 THEN 'Excellent'
        WHEN grade >= 80 THEN 'Very Good'
        WHEN grade >= 70 THEN 'Good'
        WHEN grade >= 60 THEN 'Satisfactory'
        ELSE 'Failing'
    END AS letter_grade
FROM Enrollments e
JOIN Students s ON e.student_id = s.student_id;
```

Output

```
last_name | grade | letter_grade
----------+-------+-------------
```

```
Brown     | 88.5  | Very Good
Brown     | 76.0  | Good
Davis     | 72.0  | Good
Davis     | 91.0  | Excellent
Miller    | 95.5  | Excellent
Miller    | 82.0  | Very Good
Wilson    | 55.0  | Failing
Taylor    | 97.0  | Excellent
```

4.10 Useful Functions

Common Scalar Functions

Category	Function	Description
Text	`UPPER(s)`, `LOWER(s)`	Uppercase, lowercase
	`LENGTH(s)`	Length
	`SUBSTR(s, start, len)`	Substring
	`TRIM(s)`	Remove whitespace
	`REPLACE(s, a, b)`	Replace
Number	`ABS(x)`, `ROUND(x, n)`	Absolute value, rounding
	`CEIL(x)`, `FLOOR(x)`	Ceiling, floor
Date	`CURRENT_DATE`	Current date
	`DATE('now')`	Current date (SQLite)
NULL	`COALESCE(a, b, c)`	First non-NULL value
	`NULLIF(a, b)`	NULL if $a = b$

Function Examples

```sql
SELECT UPPER(last_name) AS upper_name,
       LENGTH(first_name) AS name_len,
       COALESCE(grade, 0) AS grade_or_zero
FROM Students s
LEFT JOIN Enrollments e ON s.student_id = e.student_id;
```

4.11 INSERT, UPDATE, DELETE

Inserting Data

```sql
-- Insert a single row
INSERT INTO Students (student_id, last_name, first_name, major)
VALUES (6, 'Garcia', 'Lucy', 'CS');

-- Insert multiple rows
INSERT INTO Students (student_id, last_name, first_name, major) VALUES
```

```sql
    (7, 'White', 'Mark', 'Math'),
    (8, 'Thomas', 'Julie', 'CS');

-- Insert from a query
INSERT INTO Archived_Students
SELECT * FROM Students WHERE major = 'Math';
```

Updating and Deleting

```sql
-- Update
UPDATE Enrollments SET grade = 65.0
WHERE student_id = 4 AND course_id = 101;

-- Conditional delete
DELETE FROM Enrollments WHERE grade < 40;

-- Delete all rows
DELETE FROM Archived_Students;
-- or TRUNCATE TABLE Archived_Students;  (PostgreSQL)
```

UPDATE/DELETE Without WHERE

An `UPDATE` or `DELETE` without a `WHERE` clause affects *all* rows in the table. Always verify your `WHERE` clause before executing!

4.12 Python Integration

Complete CRUD with Python/sqlite3

```python
import sqlite3

conn = sqlite3.connect(':memory:')  # in-memory database
cur = conn.cursor()

# Create and insert
cur.executescript('''
    CREATE TABLE Students (
        id INTEGER PRIMARY KEY, last_name TEXT, first_name TEXT, major
        ↪  TEXT
    );
    INSERT INTO Students VALUES (1, 'Brown', 'Alice', 'CS');
    INSERT INTO Students VALUES (2, 'Davis', 'Bob', 'CS');
    INSERT INTO Students VALUES (3, 'Miller', 'Claire', 'Math');
''')

# Read with parameters
major = 'CS'
```

```python
cur.execute("SELECT * FROM Students WHERE major = ?", (major,))
print("CS Students:", cur.fetchall())

# Update
cur.execute("UPDATE Students SET major = ? WHERE id = ?", ('Math', 2))
conn.commit()

# Verify
cur.execute("SELECT * FROM Students ORDER BY id")
for row in cur.fetchall():
    print(row)

conn.close()
```

Output

```
CS Students: [(1, 'Brown', 'Alice', 'CS'), (2, 'Davis', 'Bob', 'CS')]
(1, 'Brown', 'Alice', 'CS')
(2, 'Davis', 'Bob', 'Math')
(3, 'Miller', 'Claire', 'Math')
```

Exercises

Exercise 4.1. Write SQL queries for the following:

1. All courses with more than 3 credits.
2. Names of students born before 2004.
3. Enrollments with a grade above 85, sorted by grade descending.
4. The top 3 students by grade.

Exercise 4.2. Explain the difference between the two queries:

```sql
SELECT DISTINCT major FROM Students;
SELECT major FROM Students GROUP BY major;
```

Exercise 4.3. Write a SQL query that displays for each enrollment: the student's name, the course title, the grade, and the corresponding letter grade (Excellent ≥ 90, Very Good ≥ 80, Good ≥ 70, Satisfactory ≥ 60, Failing otherwise).

Exercise 4.4. What are the results of the following expressions?

1. `NULL = NULL`
2. `NULL <> 5`
3. `NULL AND TRUE`

4. `NULL OR TRUE`
5. `NOT NULL`

Exercise 4.5. Write a Python program that:

1. Creates a SQLite database with the University schema tables.
2. Inserts the sample data.
3. Executes 3 different queries and displays formatted results.
4. Uses parameterized queries for all user inputs.

Chapter 5

SQL — Joins and Subqueries

The power of the relational model lies in its ability to link tables together. A student is enrolled in a department, which belongs to a university; each enrolment references a course, taught by a professor. To extract complete information—for example, "which students take Professor Smith's courses?"—one must *join* several tables. The join is the queen of SQL operations, and mastering it distinguishes the beginner from the expert. Subqueries, meanwhile, allow nesting questions within one another, offering considerable expressivity.

5.1 Introduction

Joins combine data from multiple tables by exploiting the relationships between them (foreign keys). *Subqueries* (nested queries) allow using the result of one query as input to another.

5.2 Inner Join (INNER JOIN)

Definition 5.1 (Inner Join)**.** An *inner join* combines rows from two tables where the join condition is satisfied. Rows without a match are eliminated.

Equivalent Inner Join Syntaxes

```
-- Explicit syntax (recommended)
SELECT s.last_name, c.title, e.grade
FROM Students s
INNER JOIN Enrollments e ON s.student_id = e.student_id
INNER JOIN Courses c ON e.course_id = c.course_id;

-- Implicit syntax (old style, avoid)
SELECT s.last_name, c.title, e.grade
FROM Students s, Enrollments e, Courses c
WHERE s.student_id = e.student_id
  AND e.course_id = c.course_id;
```

Output

```
last_name | title      | grade
----------+------------+------
Brown     | Databases  | 88.5
Brown     | Algorithms | 76.0
Davis     | Databases  | 72.0
Davis     | Networks   | 91.0
Miller    | Algorithms | 95.5
Miller    | Statistics | 82.0
Wilson    | Databases  | 55.0
Taylor    | Statistics | 97.0
```

Always Use Explicit JOIN Syntax

The implicit syntax (comma in `FROM`) is error-prone: forgetting a condition in `WHERE` produces an accidental Cartesian product. The `JOIN ... ON` syntax is clearer and separates join logic from filtering.

5.3 Natural Join (NATURAL JOIN)

Natural Join

```
-- Joins on columns with the same name
SELECT * FROM Students NATURAL JOIN Enrollments;
```

Danger of NATURAL JOIN

Natural join automatically determines common columns. If two tables share a column name by coincidence (e.g., `last_name` in both `Students` and `Professors`), the result will be incorrect. Always prefer explicit joins with `ON`.

5.4 Outer Joins (OUTER JOIN)

Definition 5.2 (Outer Join)**.** An *outer join* preserves all rows from one table (or both), even those without a match in the other table. Missing columns are filled with `NULL`.

5.4.1 LEFT JOIN

LEFT JOIN — All Students, Even Without Enrollments

```
SELECT s.last_name, s.first_name, c.title, e.grade
FROM Students s
LEFT JOIN Enrollments e ON s.student_id = e.student_id
LEFT JOIN Courses c ON e.course_id = c.course_id
```

```
ORDER BY s.last_name;
```

If a student is not enrolled in any course, they still appear with `NULL` for `title` and `grade`.

5.4.2 RIGHT JOIN

RIGHT JOIN — All Courses, Even Without Enrollments

```
SELECT c.title, s.last_name, e.grade
FROM Enrollments e
RIGHT JOIN Courses c ON e.course_id = c.course_id
LEFT JOIN Students s ON e.student_id = s.student_id;
```

Remark 5.3. `RIGHT JOIN` is less commonly used because it can always be rewritten as a `LEFT JOIN` by reversing the table order. SQLite does not support `RIGHT JOIN` (before version 3.39).

5.4.3 FULL OUTER JOIN

FULL OUTER JOIN

```
-- All students and all courses (PostgreSQL)
SELECT s.last_name, c.title
FROM Students s
FULL OUTER JOIN Enrollments e ON s.student_id = e.student_id
FULL OUTER JOIN Courses c ON e.course_id = c.course_id;
```

Join Types Summary

Type	Description
`INNER JOIN`	Rows with matches in both tables
`LEFT JOIN`	All rows from the left table + matches
`RIGHT JOIN`	All rows from the right table + matches
`FULL OUTER JOIN`	All rows from both tables
`CROSS JOIN`	Cartesian product (all combinations)

5.5 Cross Join (CROSS JOIN)

Cartesian Product

```
-- All student-course combinations
SELECT s.last_name, c.title
FROM Students s
```

```sql
CROSS JOIN Courses c;
-- 5 students x 4 courses = 20 rows
```

5.6 Self Join

Definition 5.4 (Self Join)**.** A *self join* is a join of a table with itself. It requires aliases to distinguish the two "copies" of the table.

Finding Pairs of Students in the Same Major

```sql
SELECT s1.last_name AS student1, s2.last_name AS student2, s1.major
FROM Students s1
JOIN Students s2 ON s1.major = s2.major
                 AND s1.student_id < s2.student_id;
```

Output

```
student1 | student2 | major
---------+----------+------
Brown    | Davis    | CS
Brown    | Wilson   | CS
Davis    | Wilson   | CS
Miller   | Taylor   | Math
```

5.7 Subqueries

Definition 5.5 (Subquery)**.** A *subquery* is a `SELECT` statement nested inside another query. It can appear in the `WHERE`, `FROM`, or `SELECT` clauses.

5.7.1 Scalar Subquery

A subquery that returns exactly one value:

Scalar Subquery

```sql
-- Students with a grade above the average
SELECT s.last_name, e.grade
FROM Students s
JOIN Enrollments e ON s.student_id = e.student_id
WHERE e.grade > (SELECT AVG(grade) FROM Enrollments);
```

Output

```
-- Average = 83.4375
last_name | grade
```

```
----------+------
Brown     | 88.5
Davis     | 91.0
Miller    | 95.5
Taylor    | 97.0
```

5.7.2 Subquery with IN

Subquery with IN

```sql
-- Students enrolled in a course taught by Prof. Smith
SELECT last_name, first_name
FROM Students
WHERE student_id IN (
    SELECT student_id
    FROM Enrollments
    WHERE course_id IN (
        SELECT course_id FROM Courses WHERE prof_id = 1
    )
);
```

5.7.3 Correlated Subquery

Definition 5.6 (Correlated Subquery)**.** A subquery is *correlated* when it references a column from the outer query. It is re-evaluated for each row of the outer query.

Correlated Subquery with EXISTS

```sql
-- Students enrolled in at least one course
SELECT s.last_name, s.first_name
FROM Students s
WHERE EXISTS (
    SELECT 1
    FROM Enrollments e
    WHERE e.student_id = s.student_id
);

-- Students NOT enrolled in any course
SELECT s.last_name, s.first_name
FROM Students s
WHERE NOT EXISTS (
    SELECT 1
    FROM Enrollments e
    WHERE e.student_id = s.student_id
);
```

Performance of Correlated Subqueries

Correlated subqueries are re-evaluated for each row. On large tables, they can be slow. Modern optimizers often transform them into joins, but it is good to know the alternative with `JOIN`:

```
-- Equivalent with LEFT JOIN + IS NULL
SELECT s.last_name FROM Students s
LEFT JOIN Enrollments e ON s.student_id = e.student_id
WHERE e.student_id IS NULL;
```

5.7.4 Subquery in FROM (Derived Table)

Subquery in FROM

```
-- Average per student, then filter
SELECT sub.last_name, sub.avg_grade
FROM (
    SELECT s.last_name, AVG(e.grade) AS avg_grade
    FROM Students s
    JOIN Enrollments e ON s.student_id = e.student_id
    GROUP BY s.student_id, s.last_name
) AS sub
WHERE sub.avg_grade > 80;
```

Output

```
last_name | avg_grade
----------+----------
Brown     | 82.25
Davis     | 81.5
Miller    | 88.75
Taylor    | 97.0
```

5.7.5 Subquery in SELECT

Subquery in SELECT

```
SELECT s.last_name,
       (SELECT COUNT(*) FROM Enrollments e
        WHERE e.student_id = s.student_id) AS num_courses
FROM Students s;
```

Output

```
last_name | num_courses
----------+------------
Brown     | 2
Davis     | 2
Miller    | 2
Wilson    | 1
Taylor    | 1
```

5.8 ALL, ANY/SOME Operators

Comparison with ALL and ANY

```sql
-- Grade higher than ALL grades in course 101
SELECT s.last_name, e.grade
FROM Students s
JOIN Enrollments e ON s.student_id = e.student_id
WHERE e.grade > ALL (
    SELECT grade FROM Enrollments WHERE course_id = 101
);

-- Grade higher than at least ONE grade in course 101
SELECT s.last_name, e.grade
FROM Students s
JOIN Enrollments e ON s.student_id = e.student_id
WHERE e.grade > ANY (
    SELECT grade FROM Enrollments WHERE course_id = 101
);
```

5.9 Common Table Expressions (CTE)

Definition 5.7 (CTE (Common Table Expression))**.** A *CTE* is a temporary named query, defined with the `WITH` keyword, that can be referenced in the main query. It improves readability and enables recursion.

Simple CTE

```sql
WITH Averages AS (
    SELECT student_id, AVG(grade) AS avg_grade
    FROM Enrollments
    GROUP BY student_id
)
SELECT s.last_name, a.avg_grade
FROM Students s
JOIN Averages a ON s.student_id = a.student_id
WHERE a.avg_grade > 80
```

```
ORDER BY a.avg_grade DESC;
```

Recursive CTE — Hierarchy

```
-- Categories table with parent reference
CREATE TABLE Categories (
    id INTEGER PRIMARY KEY,
    name TEXT,
    parent_id INTEGER REFERENCES Categories(id)
);

-- Recursive hierarchy traversal
WITH RECURSIVE Tree AS (
    -- Base case: roots
    SELECT id, name, parent_id, 0 AS depth
    FROM Categories WHERE parent_id IS NULL

    UNION ALL

    -- Recursive case
    SELECT c.id, c.name, c.parent_id, t.depth + 1
    FROM Categories c
    JOIN Tree t ON c.parent_id = t.id
)
SELECT * FROM Tree ORDER BY depth, name;
```

5.10 Python Integration

Joins and Subqueries with Python

```
import sqlite3

conn = sqlite3.connect('university.db')
cur = conn.cursor()

# Join: students and their courses
cur.execute('''
    SELECT s.last_name, c.title, e.grade
    FROM Students s
    JOIN Enrollments e ON s.student_id = e.student_id
    JOIN Courses c ON e.course_id = c.course_id
    WHERE e.grade > ?
    ORDER BY e.grade DESC
''', (80,))

print("Students with grade > 80:")
for name, course, grade in cur.fetchall():
```

```
    print(f"  {name:10} | {course:20} | {grade:.1f}")

# Subquery: students above average
cur.execute('''
    SELECT last_name, first_name FROM Students
    WHERE student_id IN (
        SELECT student_id FROM Enrollments
        WHERE grade > (SELECT AVG(grade) FROM Enrollments)
    )
''')
print("\nAbove average:", cur.fetchall())

conn.close()
```

Exercises

Exercise 5.1. Write a query that displays each professor's name and the number of courses they teach. Include professors who teach no courses.

Exercise 5.2. Find students enrolled in all courses taught by the professor with `prof_id` = 2. Use a subquery with `NOT EXISTS`.

Exercise 5.3. Write the same query as the previous exercise using `GROUP BY` and `HAVING` with `COUNT`. Compare the two approaches.

Exercise 5.4. Find pairs of students who share at least two courses. Display the names of both students and the number of shared courses.

Exercise 5.5. Using a recursive CTE, generate a table of numbers from 1 to 20 and display their factorial.

Chapter 6

SQL — Aggregation and Window Functions

Until now, our SQL queries returned individual rows. But often, we want summaries: the total number of students, the average grade by department, the revenue per quarter. This is the role of *aggregate functions* (`COUNT`, `SUM`, `AVG`, `MAX`, `MIN`) combined with `GROUP BY`. *Window functions* (`OVER`), introduced in SQL:2003, go further: they compute aggregates while preserving individual rows—rank, cumulative sum, moving average—without requiring complex subqueries.

6.1 Aggregate Functions

Definition 6.1 (Aggregate Function)**.** An *aggregate function* operates on a set of rows and returns a single value summarizing that set.

Standard Aggregate Functions

Function	Description	Ignores NULL?
`COUNT(*)`	Number of rows	No
`COUNT(col)`	Number of non-NULL values	Yes
`COUNT(DISTINCT col)`	Number of distinct values	Yes
`SUM(col)`	Sum	Yes
`AVG(col)`	Average	Yes
`MIN(col)`	Minimum	Yes
`MAX(col)`	Maximum	Yes

Simple Aggregations

```
SELECT COUNT(*) AS num_enrollments,
       COUNT(DISTINCT student_id) AS num_students,
       AVG(grade) AS avg_grade,
       MIN(grade) AS min_grade,
       MAX(grade) AS max_grade,
       SUM(grade) AS total_grades
FROM Enrollments;
```

Output

```
num_enrollments | num_students | avg_grade | min_grade | max_grade | total_grades
----------------+--------------+-----------+-----------+-----------+-------------
8               | 5            | 83.4375   | 55.0      | 97.0      | 667.5
```

AVG and NULL Values

`AVG` ignores `NULL` values. The average of $\{70, \texttt{NULL}, 90\}$ is 80, not 53.3. If you want to treat NULLs as 0, use `AVG(COALESCE(grade, 0))`.

6.2 GROUP BY — Grouping

Definition 6.2 (GROUP BY)**.** The `GROUP BY` clause partitions rows into groups by one or more columns. Aggregate functions are then applied to each group separately.

Execution Order with GROUP BY

1. `FROM` / `JOIN`
2. `WHERE` (filter before grouping)
3. `GROUP BY`
4. `HAVING` (filter after grouping)
5. `SELECT`
6. `DISTINCT`
7. `ORDER BY`
8. `LIMIT`

Average per Student

```sql
SELECT s.last_name, s.first_name,
       COUNT(*) AS num_courses,
       ROUND(AVG(e.grade), 2) AS avg_grade,
       MIN(e.grade) AS min_grade,
       MAX(e.grade) AS max_grade
FROM Students s
JOIN Enrollments e ON s.student_id = e.student_id
GROUP BY s.student_id, s.last_name, s.first_name
ORDER BY avg_grade DESC;
```

Output

```
last_name | first_name | num_courses | avg_grade | min_grade | max_grade
----------+------------+-------------+-----------+-----------+----------
```

```
Taylor    | Emma      | 1          | 97.0       | 97.0       | 97.0
Miller    | Claire    | 2          | 88.75      | 82.0       | 95.5
Brown     | Alice     | 2          | 82.25      | 76.0       | 88.5
Davis     | Bob       | 2          | 81.5       | 72.0       | 91.0
Wilson    | David     | 1          | 55.0       | 55.0       | 55.0
```

Column Outside GROUP BY Without Aggregation

Every column in the `SELECT` must either appear in `GROUP BY` or be inside an aggregate function:

```sql
-- ERROR: first_name not in GROUP BY or aggregated
SELECT last_name, first_name, AVG(grade) FROM Enrollments
JOIN Students ON ... GROUP BY last_name;

-- CORRECT:
SELECT last_name, first_name, AVG(grade) FROM Enrollments
JOIN Students ON ... GROUP BY last_name, first_name;
```

MySQL in non-strict mode silently accepts this error (indeterminate result). PostgreSQL and SQLite correctly reject it.

6.3 HAVING — Filtering After Grouping

Definition 6.3 (HAVING). The `HAVING` clause filters groups after aggregation. It is to `GROUP BY` what `WHERE` is to `FROM`: `WHERE` filters rows, `HAVING` filters groups.

Filtering with HAVING

```sql
-- Students with average above 80
SELECT s.last_name, AVG(e.grade) AS avg_grade
FROM Students s
JOIN Enrollments e ON s.student_id = e.student_id
GROUP BY s.student_id, s.last_name
HAVING AVG(e.grade) > 80
ORDER BY avg_grade DESC;

-- Courses with at least 2 enrolled students
SELECT c.title, COUNT(*) AS num_enrolled
FROM Courses c
JOIN Enrollments e ON c.course_id = e.course_id
GROUP BY c.course_id, c.title
HAVING COUNT(*) >= 2;
```

Output

```
-- Average > 80:
last_name | avg_grade
----------+----------
Taylor    | 97.0
Miller    | 88.75
Brown     | 82.25
Davis     | 81.5

-- Courses with >= 2 enrolled:
title      | num_enrolled
-----------+-------------
Databases  | 3
Algorithms | 2
Statistics | 2
```

6.4 Advanced Grouping

6.4.1 GROUPING SETS, ROLLUP, CUBE

ROLLUP — Hierarchical Subtotals (PostgreSQL)

```sql
SELECT s.major,
       c.title,
       AVG(e.grade) AS avg_grade,
       COUNT(*) AS num
FROM Students s
JOIN Enrollments e ON s.student_id = e.student_id
JOIN Courses c ON e.course_id = c.course_id
GROUP BY ROLLUP (s.major, c.title)
ORDER BY s.major, c.title;
```

`ROLLUP` generates hierarchical subtotals: by (major, course), by major, and the grand total.

`CUBE` generates all possible combinations of subtotals.

6.5 Window Functions

Definition 6.4 (Window Function)**.** A *window function* performs a calculation across a set of rows related to the current row, *without reducing the number of rows* in the result. It uses the `OVER()` clause.

Theorem 6.5 (GROUP BY vs. Window Function)**.**

- *`GROUP BY` + aggregation: reduces rows (one row per group).*
- *Window function: keeps all rows while adding a computed value.*

6.5.1 OVER Syntax

Window Function Syntax

```
function(args) OVER (
    [PARTITION BY columns]
    [ORDER BY columns [ASC|DESC]]
    [ROWS BETWEEN start AND end]
)
```

6.5.2 Windowed Aggregation

Windowed Average by Major

```
SELECT s.last_name, s.major, e.grade,
       AVG(e.grade) OVER (PARTITION BY s.major) AS major_avg,
       e.grade - AVG(e.grade) OVER (PARTITION BY s.major) AS deviation
FROM Students s
JOIN Enrollments e ON s.student_id = e.student_id
ORDER BY s.major, s.last_name;
```

Output

```
last_name | major | grade | major_avg | deviation
----------+-------+-------+-----------+----------
Brown     | CS    | 88.5  | 76.5      | 12.0
Brown     | CS    | 76.0  | 76.5      | -0.5
Davis     | CS    | 72.0  | 76.5      | -4.5
Davis     | CS    | 91.0  | 76.5      | 14.5
Wilson    | CS    | 55.0  | 76.5      | -21.5
Miller    | Math  | 95.5  | 91.5      | 4.0
Miller    | Math  | 82.0  | 91.5      | -9.5
Taylor    | Math  | 97.0  | 91.5      | 5.5
```

6.5.3 ROW_NUMBER, RANK, DENSE_RANK

Definition 6.6 (Ranking Functions).

- `ROW_NUMBER()` — unique sequential number within the partition.
- `RANK()` — rank with gaps (ties receive the same rank; the next rank is incremented by the number of ties).
- `DENSE_RANK()` — rank without gaps.

Ranking Students by Grade

```
SELECT s.last_name, e.grade,
       ROW_NUMBER() OVER (ORDER BY e.grade DESC) AS row_num,
```

```
       RANK()       OVER (ORDER BY e.grade DESC) AS rnk,
       DENSE_RANK() OVER (ORDER BY e.grade DESC) AS dense_rnk
FROM Students s
JOIN Enrollments e ON s.student_id = e.student_id;
```

Output

```
last_name | grade | row_num | rnk | dense_rnk
----------+-------+---------+-----+----------
Taylor    | 97.0  | 1       | 1   | 1
Miller    | 95.5  | 2       | 2   | 2
Davis     | 91.0  | 3       | 3   | 3
Brown     | 88.5  | 4       | 4   | 4
Miller    | 82.0  | 5       | 5   | 5
Brown     | 76.0  | 6       | 6   | 6
Davis     | 72.0  | 7       | 7   | 7
Wilson    | 55.0  | 8       | 8   | 8
```

Top Student per Major (Top-N per Group)

```
WITH Ranked AS (
    SELECT s.last_name, s.major, e.grade,
           ROW_NUMBER() OVER (
               PARTITION BY s.major
               ORDER BY e.grade DESC
           ) AS rn
    FROM Students s
    JOIN Enrollments e ON s.student_id = e.student_id
)
SELECT last_name, major, grade
FROM Ranked
WHERE rn = 1;
```

6.5.4 LAG, LEAD

Definition 6.7 (LAG and LEAD).

- `LAG(col, n, default)` — value of `col` n rows *before* the current row.
- `LEAD(col, n, default)` — value of `col` n rows *after* the current row.

Comparing with the Previous Row

```
SELECT s.last_name, e.grade,
       LAG(e.grade, 1) OVER (ORDER BY e.grade DESC) AS prev_grade,
       e.grade - LAG(e.grade, 1) OVER (ORDER BY e.grade DESC) AS diff
FROM Students s
JOIN Enrollments e ON s.student_id = e.student_id
```

```
ORDER BY e.grade DESC;
```

6.5.5 Sliding Windows (Frame)

Moving Average over 3 Rows

```
SELECT s.last_name, e.grade,
       AVG(e.grade) OVER (
           ORDER BY e.grade
           ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING
       ) AS moving_avg
FROM Students s
JOIN Enrollments e ON s.student_id = e.student_id
ORDER BY e.grade;
```

Window Frame Specifications

Specification	Description
`ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`	From start to current row
`ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING`	Sliding window of 3 rows
`ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING`	From current row to end
`RANGE BETWEEN ...`	Value-based (not position-based)

6.5.6 NTILE and Cumulative Percentages

Quartiles and Cumulative Percentage

```
SELECT s.last_name, e.grade,
       NTILE(4) OVER (ORDER BY e.grade DESC) AS quartile,
       ROUND(100.0 * SUM(e.grade) OVER (ORDER BY e.grade DESC
           ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
           / SUM(e.grade) OVER (), 1) AS cumul_pct
FROM Students s
JOIN Enrollments e ON s.student_id = e.student_id
ORDER BY e.grade DESC;
```

6.6 Python Integration

Aggregation and Window Functions with Python

```python
import sqlite3

conn = sqlite3.connect('university.db')
cur = conn.cursor()

# Aggregation with GROUP BY
cur.execute('''
    SELECT s.last_name, COUNT(*) as num, ROUND(AVG(e.grade), 2) as avg
    FROM Students s
    JOIN Enrollments e ON s.student_id = e.student_id
    GROUP BY s.student_id, s.last_name
    HAVING AVG(e.grade) > 80
    ORDER BY avg DESC
''')
print("Averages > 80:")
for row in cur.fetchall():
    print(f"  {row[0]:10} | {row[1]} courses | avg {row[2]}")

# Window function (SQLite >= 3.25)
cur.execute('''
    SELECT s.last_name, e.grade,
           RANK() OVER (ORDER BY e.grade DESC) as rnk
    FROM Students s
    JOIN Enrollments e ON s.student_id = e.student_id
''')
print("\nRankings:")
for name, grade, rnk in cur.fetchall():
    print(f"  #{rnk} {name:10} : {grade}")

conn.close()
```

Exercises

Exercise 6.1. For each course, display the number of enrolled students, the average grade, the minimum, and the maximum. Only show courses with at least 2 students.

Exercise 6.2. Rank students by their overall average and display their rank. Use `DENSE_RANK` to handle ties.

Exercise 6.3. For each enrollment, display the student's grade, their major's average, and the deviation from that average. Use a window function.

Exercise 6.4. Display the top 2 grades per course. Use `ROW_NUMBER` with `PARTITION BY`.

Exercise 6.5. Calculate the cumulative sum of grades per student, ordered by course ID, using a sliding window.

Chapter 7

Database Normalization

Why does a well-designed database never contain redundancy? The answer lies in a single word: *anomalies*. When the same information is stored in multiple places, a partial update creates inconsistencies; a deletion can destroy data one intended to keep; an insertion may be impossible without inventing fictitious values. Edgar Codd, as early as 1972, formalized these problems by introducing *functional dependencies* and *normal forms*. The idea is systematic: decompose tables to eliminate redundancy while preserving information (lossless decomposition) and constraints (dependency preservation). First normal form (1NF), second (2NF), third (3NF), and Boyce–Codd normal form (BCNF) form an increasingly strict hierarchy. Understanding this hierarchy is understanding the art of designing robust relational schemas.

7.1 Design Problems and Anomalies

A poorly designed schema leads to *anomalies*:

Definition 7.1 (Data Anomalies)**.**

- **Insertion anomaly**: inability to insert data without also inserting other unwanted data.
- **Update anomaly**: a modification must be repeated across multiple rows to maintain consistency.
- **Deletion anomaly**: deleting one piece of information causes the unintended loss of other information.

Example 7.2. Consider the following unnormalized table:

s_id	name	course	prof	dept	grade
1	Brown	DB	Smith	CS	88.5
1	Brown	Algo	Johnson	Math	76.0
2	Davis	DB	Smith	CS	72.0

- **Update**: if Smith changes department, *all* rows where Smith appears must be modified.
- **Insertion**: we cannot record a new course without enrolling a student.
- **Deletion**: deleting Davis's DB enrollment may lose course info if it is the last row.

7.2 Functional Dependencies

Definition 7.3 (Functional Dependency)**.** Let R be a relation and $X, Y \subseteq \text{attr}(R)$. We say Y *functionally depends* on X, denoted $X \to Y$, if for all pairs of tuples $t_1, t_2 \in R$:

$$t_1[X] = t_2[X] \implies t_1[Y] = t_2[Y]$$

In other words, the value of X uniquely determines the value of Y.

Example 7.4. In the University database:

- student_id $\to$ last_name, first_name, major
- course_id $\to$ title, credits, prof_id
- prof_id $\to$ prof_name, department
- (student_id, course_id) $\to$ grade

Definition 7.5 (Trivial Functional Dependency)**.** $X \to Y$ is *trivial* if $Y \subseteq X$.

Definition 7.6 (Partial and Transitive Dependencies)**.**

- **Partial**: $X \to Y$ is partial if there exists $X' \subsetneq X$ such that $X' \to Y$.
- **Transitive**: if $X \to Z$ and $Z \to Y$ (where Z is not a key), then $X \to Y$ is transitive.

7.3 Armstrong's Axioms

Theorem 7.7 (Armstrong's Axioms)**.** *Armstrong's axioms are* sound *and* complete*:*

1. ***Reflexivity****: if $Y \subseteq X$, then $X \to Y$.*
2. ***Augmentation****: if $X \to Y$, then $XZ \to YZ$.*
3. ***Transitivity****: if $X \to Y$ and $Y \to Z$, then $X \to Z$.*

Derived rules:

- ***Union****: if $X \to Y$ and $X \to Z$, then $X \to YZ$.*
- ***Decomposition****: if $X \to YZ$, then $X \to Y$ and $X \to Z$.*
- ***Pseudotransitivity****: if $X \to Y$ and $WY \to Z$, then $WX \to Z$.*

7.4 Attribute Closure

Definition 7.8 (Closure X^+)**.** The *closure* of X with respect to a set of FDs F, denoted X_F^+, is the set of all attributes A such that $X \to A$ can be derived from F using Armstrong's axioms.

Proposition 7.9 (Closure Algorithm)**.** To compute X^+:

1. Initialize $X^+ = X$.

2. Repeat: for each FD $Y \to Z$ in F, if $Y \subseteq X^+$, add Z to X^+.

3. Stop when X^+ no longer changes.

Example 7.10. Let $R(A, B, C, D, E)$ with $F = \{A \to B,\ BC \to D,\ D \to E\}$.
Compute $\{A, C\}^+$:

1. $X^+ = \{A, C\}$

2. $A \to B$ and $A \in X^+$: $X^+ = \{A, B, C\}$

3. $BC \to D$ and $\{B, C\} \subseteq X^+$: $X^+ = \{A, B, C, D\}$

4. $D \to E$ and $D \in X^+$: $X^+ = \{A, B, C, D, E\}$

Since X^+ contains all attributes, $\{A, C\}$ is a candidate key.

7.5 First Normal Form (1NF)

Definition 7.11 (1NF)**.** A relation is in *first normal form* (1NF) if all its attributes are *atomic*: each cell contains a single, indivisible value.

Example 7.12. Table NOT in 1NF:

id	name	phones
1	Brown	555-1234, 555-5678

Table in 1NF (after decomposition):

id	name
1	Brown

id	phone
1	555-1234
1	555-5678

7.6 Second Normal Form (2NF)

Definition 7.13 (2NF)**.** A relation is in *second normal form* if it is in 1NF and every non-key attribute *fully* depends on the primary key (no partial functional dependency).

Remark 7.14. 2NF is only relevant for relations with a composite key. If the key is a single attribute, a relation in 1NF is automatically in 2NF.

Example 7.15. Let $R(\underline{\text{s\_id}}, \underline{\text{c\_id}}, \text{s\_name}, \text{grade})$ with key $(\text{s\_id}, \text{c\_id})$.
s_id $\to$ s_name is a partial FD (depends on only part of the key). Not in 2NF.
Decomposition:

- $R_1(\underline{\text{s\_id}}, \text{s\_name})$
- $R_2(\underline{\text{s\_id}}, \underline{\text{c\_id}}, \text{grade})$

7.7 Third Normal Form (3NF)

Definition 7.16 (3NF)**.** A relation is in *third normal form* if it is in 2NF and no non-key attribute transitively depends on the primary key.

Formally, for every non-trivial FD $X \to A$:

- either X contains a candidate key (superkey),
- or A is a prime attribute (belongs to some candidate key).

Example 7.17. Let $R(\underline{\text{s\_id}}, \text{major}, \text{major\_advisor})$.
FDs: s_id $\to$ major and major $\to$ major_advisor.
The FD s_id $\to$ major_advisor is transitive. Not in 3NF.
Decomposition:

- $R_1(\underline{\text{s\_id}}, \text{major})$
- $R_2(\underline{\text{major}}, \text{major\_advisor})$

7.8 Boyce-Codd Normal Form (BCNF)

Definition 7.18 (BCNF)**.** A relation is in *BCNF* if, for every non-trivial functional dependency $X \to Y$, X is a superkey.

Theorem 7.19 (BCNF vs. 3NF)**.**

- *BCNF* $\Rightarrow$ *3NF* $\Rightarrow$ *2NF* $\Rightarrow$ *1NF.*
- *BCNF is stricter than 3NF: it forbids even dependencies where the determinant is not a superkey, even if the dependent is a prime attribute.*
- *Decomposition into BCNF does not always preserve FDs.*
- *Decomposition into 3NF always preserves FDs.*

Example 7.20. Let $R(\underline{A}, \underline{B}, C)$ with FDs: $AB \to C$ and $C \to A$.
$C \to A$: C is not a superkey (the keys are $\{A, B\}$ and $\{B, C\}$). Not in BCNF.
But A is prime (part of key $\{A, B\}$). So it is in 3NF.

7.9 Lossless Decomposition

Theorem 7.21 (Lossless Join Decomposition)**.** *A decomposition of R into R_1 and R_2 is* lossless *if and only if one of the following FDs holds:*

$$R_1 \cap R_2 \to R_1 - R_2 \quad \textit{or} \quad R_1 \cap R_2 \to R_2 - R_1$$

i.e., the intersection of the schemas determines at least one of the complements.

7.10 BCNF Decomposition Algorithm

Proposition 7.22 (BCNF Algorithm)**.** 1. Check if the relation is in BCNF.

2. If not, find an FD $X \to Y$ violating BCNF (where X is not a superkey).

3. Decompose into:

   - $R_1 = X \cup Y$ (with key X)
   - $R_2 = R - Y \cup X$ (all attributes except Y, but including X)

4. Repeat on R_1 and R_2 if necessary.

7.11 Normal Forms Summary

Normal Forms Overview

NF	Condition
1NF	All attributes are atomic
2NF	1NF + no partial FD of a non-key attribute on the key
3NF	2NF + no transitive FD of a non-key attribute on the key
BCNF	For every non-trivial FD $X \to Y$, X is a superkey

7.12 3NF Synthesis (Bernstein's Algorithm)

Proposition 7.23 (3NF Synthesis Algorithm)**.** 1. Compute a minimal cover $F_{\min}$ of the FDs.

2. For each FD $X \to Y$ in $F_{\min}$, create a relation $R_i(X, Y)$.

3. If no relation contains a candidate key of R, add a relation with a candidate key.

4. Eliminate redundant relations (those included in another).

This algorithm guarantees:

- Lossless decomposition.
- Preservation of functional dependencies.
- Result in 3NF.

7.13 Complete Normalization Example

Unnormalized Table

```
CREATE TABLE Enrollments_Denorm (
    student_id INTEGER,
    student_name TEXT,
    major TEXT,
    major_advisor TEXT,
    course_id INTEGER,
    course_title TEXT,
    prof_id INTEGER,
    prof_name TEXT,
    grade REAL
);
```

Identified FDs:

- student_id $\rightarrow$ student_name, major
- major $\rightarrow$ major_advisor
- course_id $\rightarrow$ course_title, prof_id
- prof_id $\rightarrow$ prof_name
- (student_id, course_id) $\rightarrow$ grade

Result After 3NF/BCNF Normalization

```
CREATE TABLE Students (
    student_id INTEGER PRIMARY KEY,
    last_name TEXT NOT NULL,
    major TEXT REFERENCES Majors(major)
);

CREATE TABLE Majors (
    major TEXT PRIMARY KEY,
    advisor TEXT NOT NULL
);

CREATE TABLE Professors (
    prof_id INTEGER PRIMARY KEY,
    last_name TEXT NOT NULL
);

CREATE TABLE Courses (
    course_id INTEGER PRIMARY KEY,
    title TEXT NOT NULL,
    prof_id INTEGER REFERENCES Professors(prof_id)
);
```

```sql
CREATE TABLE Enrollments (
    student_id INTEGER REFERENCES Students(student_id),
    course_id INTEGER REFERENCES Courses(course_id),
    grade REAL,
    PRIMARY KEY (student_id, course_id)
);
```

7.14 Controlled Denormalization

Remark 7.24. Perfect normalization is not always the best choice in practice. *Denormalization* consists of intentionally reintroducing redundancy to improve read performance (avoiding costly joins).

When to Denormalize?

- When reads are much more frequent than writes.
- For reporting tables or data warehouses.
- Always document and justify denormalization.
- Use triggers or materialized views to maintain consistency.

Exercises

Exercise 7.1. Let $R(A, B, C, D, E)$ with $F = \{A \to BC,\ C \to D,\ BD \to E\}$.

1. Compute $\{A\}^+$.
2. Identify the candidate keys.
3. Is the relation in 2NF? 3NF? BCNF? Justify.
4. Decompose into BCNF.

Exercise 7.2. Normalize the following relation into 3NF using the synthesis algorithm:
R(OrderNum, OrderDate, CustNum, CustName, ProdNum, ProdName, Quantity, UnitPrice)
FDs: OrderNum $\to$ OrderDate, CustNum; CustNum $\to$ CustName; ProdNum $\to$ ProdName, UnitPrice; (OrderNum, ProdNum) $\to$ Quantity.

Exercise 7.3. Give an example of a relation that is in 3NF but not in BCNF. Decompose it into BCNF and verify whether the functional dependencies are preserved.

Exercise 7.4. Verify that the decomposition of $R(A, B, C)$ with $F = \{A \to B\}$ into $R_1(A, B)$ and $R_2(A, C)$ is lossless. Use Heath's theorem.

Chapter 8

Transactions and Concurrency Control

Imagine a bank transfer: one account is debited and another is credited. What happens if the system crashes between the two operations? The money has vanished. It was to prevent such catastrophes that Jim Gray, in the 1970s at IBM, formalized the notion of a *transaction* and the ACID properties (Atomicity, Consistency, Isolation, Durability). A transaction is a logical "all or nothing": either all its operations succeed, or none takes effect. When multiple transactions execute concurrently, concurrency control (locking, timestamping, MVCC) guarantees that the result is equivalent to a serial execution. Gray received the Turing Award in 1998 for this foundational work.

Central idea

A transaction is a logical unit of work that must be executed atomically, consistently, in isolation, and durably (ACID). Concurrency control ensures that the simultaneous execution of multiple transactions produces a correct result, as if they had been executed sequentially.

8.1 The concept of a transaction

Definition 8.1 (Transaction)**.** A **transaction** is a sequence of read and write operations on the database, delimited by a `BEGIN` and terminated by a `COMMIT` (validation) or a `ROLLBACK` (abort).

```python
# Example in Python with psycopg2
import psycopg2

conn = psycopg2.connect("dbname=bank")
cur = conn.cursor()
try:
    cur.execute("UPDATE accounts SET balance = balance - 100 WHERE id = 1")
    cur.execute("UPDATE accounts SET balance = balance + 100 WHERE id = 2")
    conn.commit()   # Both operations are committed together
except Exception:
    conn.rollback()  # No modification is applied
```

```
finally:
    cur.close()
    conn.close()
```

8.2 ACID properties

Definition 8.2 (ACID properties)**.** The four fundamental properties of a transaction are:

1. **Atomicity** — All operations are executed or none.
2. **Consistency** — The transaction brings the database from one consistent state to another.
3. **Isolation** — Concurrent transactions do not see each other's intermediate modifications.
4. **Durability** — Once committed, modifications survive any crash.

Remark 8.3. Atomicity and durability are ensured by the *Write-Ahead Log* (WAL). Consistency is ensured by integrity constraints. Isolation is ensured by concurrency control mechanisms.

8.3 Concurrency anomalies

Without control, concurrent execution can produce anomalies:

Definition 8.4 (Concurrency anomalies)**.**

- **Dirty read** — T_2 reads data modified by T_1 that has not yet committed.
- **Non-repeatable read** — T_1 reads the same data twice and gets different values because T_2 modified it in between.
- **Phantom read** — T_1 executes a query twice and gets different rows because T_2 inserted or deleted rows.
- **Lost update** — Two transactions read the same value, modify it, and write it back: one modification is lost.

8.4 Isolation levels

Definition 8.5 (SQL isolation levels)**.** The SQL standard defines four isolation levels:

Level	Dirty read	Non-repeatable	Phantom
`READ UNCOMMITTED`	Possible	Possible	Possible
`READ COMMITTED`	No	Possible	Possible
`REPEATABLE READ`	No	No	Possible
`SERIALIZABLE`	No	No	No

Attention

The default level varies by DBMS: `READ COMMITTED` for PostgreSQL and Oracle, `REPEATABLE READ` for MySQL/InnoDB. Choosing too weak a level exposes you to anomalies; too strong a level reduces performance.

```
-- Setting the isolation level in SQL
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN;
    SELECT balance FROM accounts WHERE id = 1;
    UPDATE accounts SET balance = balance - 100 WHERE id = 1;
COMMIT;
```

8.5 Serializability theory

Definition 8.6 (Schedule)**.** A **schedule** S is a sequence of operations (reads, writes, commits, rollbacks) from multiple transactions that preserves the internal order of each transaction.

Definition 8.7 (Conflict serializability)**.** Two operations **conflict** if they operate on the same data item and at least one is a write. A schedule S is **conflict-serializable** if it is conflict-equivalent to some serial schedule.

Theorem 8.8 (Precedence graph test)**.** *Construct the* ***precedence graph*** $G = (V, E)$ *where* V *is the set of transactions and* $(T_i, T_j) \in E$ *if an operation of* T_i *conflicts with an operation of* T_j *and precedes it. The schedule is conflict-serializable if and only if* G *is acyclic.*

Example 8.9. Let $S : r_1(A), w_1(A), r_2(A), w_2(A), r_1(B), w_1(B), r_2(B), w_2(B)$. Conflicts: $w_1(A) \to r_2(A)$, $w_1(A) \to w_2(A)$, $w_1(B) \to r_2(B)$, $w_1(B) \to w_2(B)$. The graph has a single edge $T_1 \to T_2$, so S is serializable (equivalent to T_1, T_2).

8.6 Two-phase locking (2PL)

Definition 8.10 (2PL protocol)**.** The **two-phase locking** (2PL) protocol requires each transaction to go through two phases:

1. **Growing phase**: the transaction acquires locks (shared for reads, exclusive for writes) without releasing any.
2. **Shrinking phase**: the transaction releases locks without acquiring new ones.

Theorem 8.11 (Correctness of 2PL)**.** *Every schedule produced by the 2PL protocol is conflict-serializable.*

Definition 8.12 (Strict and rigorous 2PL)**.**

- **Strict 2PL**: exclusive locks are released only at `COMMIT`/`ROLLBACK`. Avoids cascading aborts.
- **Rigorous 2PL**: *all* locks are released at `COMMIT`/`ROLLBACK`. Most commonly used mode.

8.7 Deadlock detection and prevention

Definition 8.13 (Deadlock)**.** A **deadlock** occurs when two or more transactions wait for each other, each holding a lock requested by the other.

Proposition 8.14 (Wait-for graph detection)**.** Construct the **wait-for graph**: an edge (T_i, T_j) means T_i is waiting for a lock held by T_j. A deadlock exists if and only if the graph contains a cycle.

Resolution strategies:

- **Periodic detection**: check for cycles and abort the least costly transaction (victim selection).
- **Timestamp-based prevention**: Wait-Die or Wound-Wait schemes.
- **Timeout**: abort any transaction that waits too long.

8.8 Multi-Version Concurrency Control (MVCC)

Definition 8.15 (MVCC)**. MVCC** (Multi-Version Concurrency Control) maintains multiple versions of each row. Reads never block writes and vice versa:

- Each transaction sees a consistent snapshot of the database at its start time.
- Writes create new versions without overwriting old ones.
- A garbage collector (vacuum) removes obsolete versions.

Remark 8.16. PostgreSQL uses MVCC for all isolation levels. MySQL/InnoDB uses MVCC for `READ COMMITTED` and `REPEATABLE READ`, adding additional locks for `SERIALIZABLE`.

Example 8.17. Under MVCC, transaction T_1 can read version v_1 of a row while T_2 writes version v_2: no locking is needed for the read.

```
-- PostgreSQL: observing MVCC
BEGIN;
SELECT xmin, xmax, * FROM accounts WHERE id = 1;
-- xmin = ID of the transaction that created this version
-- xmax = ID of the transaction that deleted it (0 if active)
COMMIT;
```

8.9 Write-Ahead Logging (WAL)

Definition 8.18 (Write-Ahead Logging)**.** The **WAL** (Write-Ahead Log) is a sequential log where every modification is recorded *before* being applied to the data pages. In case of a crash, the log enables:

- **REDO**: replay modifications of committed transactions.
- **UNDO**: reverse modifications of uncommitted transactions.

8.10 Key formulas

Key Formulas

- **ACID**: Atomicity, Consistency, Isolation, Durability
- **Serializability**: acyclic precedence graph $\Leftrightarrow$ serializable
- **2PL**: growing phase $\rightarrow$ lock point $\rightarrow$ shrinking phase
- **Deadlock**: cycle in the wait-for graph
- **MVCC**: non-blocking reads via multi-version snapshots

8.11 Exercises

Exercise 8.1. Given the schedule $S : r_1(A)$, $r_2(B)$, $w_1(B)$, $w_2(A)$, construct the precedence graph. Is S serializable?

Exercise 8.2. Give an example of a deadlock between two transactions operating on two bank accounts. Propose a solution using a fixed lock acquisition order.

Exercise 8.3. Write a Python script demonstrating a phantom read under `READ COMMITTED` in PostgreSQL, then show that it disappears under `SERIALIZABLE`.

Exercise 8.4. Explain why strict 2PL prevents cascading aborts. Give a counterexample with basic 2PL.

Exercise 8.5. In an MVCC system, transaction T_1 reads row r (version v_1), then T_2 modifies r and commits (version v_2), then T_1 re-reads r. Which version does T_1 see under `READ COMMITTED`? Under `REPEATABLE READ`? Justify.

Chapter 9

Indexing and Data Structures

Imagine a library of ten million books without a catalogue. To find a volume, one would need to walk through every shelf, one book at a time. This is exactly what a DBMS does during a *full table scan*: it examines every row of the table. An *index* is the database's catalogue: an auxiliary structure that locates relevant rows in $O(\log n)$ instead of $O(n)$. The B+ tree, invented by Rudolf Bayer and Edward McCreight in 1972, is the dominant indexing structure: balanced, efficient for equality and range searches, and optimized for disk access. Hash indexes, bitmap indexes, and spatial indexes (R-tree) complete the arsenal. This chapter develops these structures and the indexing strategies that make the difference between a query in milliseconds and one in minutes.

Central idea

An index is an auxiliary data structure that accelerates lookups in a table by avoiding a full table scan. Choosing the right index type and the right columns to index is essential for performance.

9.1 Motivation

Without an index, any lookup in a table of n rows requires a sequential scan in $O(n)$. A well-chosen index reduces the complexity to $O(\log n)$ or even $O(1)$.

Remark 9.1. An index speeds up reads but slows down writes (inserts, updates, deletes) because the structure must be maintained. The read/write trade-off guides index selection.

9.2 B-trees and B$^+$-trees

Definition 9.2 (B-tree of order m)**.** A **B-tree** of order m is a balanced search tree where each internal node has at most m children and at least $\lceil m/2 \rceil$ children (except the root). Keys are sorted within each node and data is stored in all nodes.

Definition 9.3 (B$^+$-tree)**.** A **B$^+$-tree** is a variant where:

- data is stored only in the **leaves**,
- internal nodes contain only routing keys,

- leaves are linked in a doubly linked list for sequential scans.

Proposition 9.4 (B^+-tree complexity)**.** For a B^+-tree of order m containing n keys:

- Height: $h = O(\log_m n)$.
- Search, insertion, deletion: $O(\log_m n)$ disk accesses.
- Range scan of k keys: $O(\log_m n + k/B)$ where B is the number of keys per leaf.

Attention

The B^+-tree is the default indexing structure in most relational DBMS (PostgreSQL, MySQL/InnoDB, Oracle, SQL Server). Do not confuse it with a binary search tree (BST), which is not suited for disk-based access.

```
-- Creating a B+ index (implicit) in SQL
CREATE INDEX idx_name ON students (last_name);

-- Composite index
CREATE INDEX idx_name_first ON students (last_name, first_name);

-- Verify index usage
EXPLAIN ANALYZE SELECT * FROM students WHERE last_name = 'Smith';
```

9.3 Hash indexes

Definition 9.5 (Hash index)**.** A **hash index** uses a function $h : \text{key} \to \{0, 1, \ldots, B-1\}$ to map a key directly to a bucket. Equality lookups run in $O(1)$ on average.

Remark 9.6. Hash indexes do **not** support range queries (`WHERE x BETWEEN a AND b`) or ordering. They are suitable only for exact equality lookups.

Definition 9.7 (Extendible hashing)**. Extendible hashing** uses a directory of pointers indexed by the first d bits of the hash. When a bucket overflows, the directory is doubled (or the bucket is split) without reorganizing the entire table.

9.4 Bitmap indexes

Definition 9.8 (Bitmap index)**.** A **bitmap index** associates each distinct value v of a column with a bit vector B_v of length n (number of rows): $B_v[i] = 1$ if row i has value v, 0 otherwise.

Proposition 9.9 (Efficiency of logical operations)**.** Queries combining multiple predicates translate into fast bitwise operations:

- `WHERE color = 'red' AND size = 'L'` $\to B_{\text{red}} \wedge B_{\text{L}}$.
- `WHERE color = 'red' OR size = 'L'` $\to B_{\text{red}} \vee B_{\text{L}}$.

Counting is done via `POPCOUNT` in $O(n/w)$ where w is the machine word size.

Remark 9.10. Bitmap indexes are particularly effective for **low-cardinality** columns (gender, status, category). They are used extensively in data warehouses (Oracle, Vertica).

9.5 Spatial indexes: R-trees

Definition 9.11 (R-tree)**.** An **R-tree** is a balanced tree for indexing multidimensional spatial data. Each node contains a **minimum bounding rectangle** (MBR) that encloses all objects in its subtree.

Example 9.12. Spatial search: "Find all restaurants within 500m":

```
-- PostGIS (spatial extension for PostgreSQL)
CREATE INDEX idx_geom ON restaurants USING GIST (geom);

SELECT name, ST_Distance(geom, ST_MakePoint(-73.99, 40.75)::geography) AS dist
FROM restaurants
WHERE ST_DWithin(geom, ST_MakePoint(-73.99, 40.75)::geography, 500)
ORDER BY dist;
```

9.6 Full-text indexes

Definition 9.13 (Inverted index)**.** An **inverted index** maps each term (word) to the list of documents (rows) that contain it, optionally with position and frequency information.

Example 9.14. PostgreSQL GIN index for full-text search

```
CREATE INDEX idx_fts ON articles USING GIN (to_tsvector('english', content));

SELECT title
FROM articles
WHERE to_tsvector('english', content) @@ to_tsquery('english', 'bayesian &
↪  statistics');
```

9.7 Index selection strategies

Definition 9.15 (Index selection problem)**.** Given a workload (set of queries with their frequencies) and a disk space budget, choose the set of indexes that minimizes total execution cost.

Heuristic rules:

1. Index columns appearing in `WHERE`, `JOIN`, and `ORDER BY` clauses.
2. Prefer covering composite indexes to avoid table access.
3. Avoid indexing frequently modified high-cardinality columns.
4. Use `EXPLAIN ANALYZE` to verify actual index usage.

Example 9.16. Covering index: all columns in the query are in the index

```
CREATE INDEX idx_covering ON orders (customer_id, order_date)
    INCLUDE (amount);
```

```
-- The following query can be satisfied without accessing the table
SELECT order_date, amount
FROM orders
WHERE customer_id = 42
ORDER BY order_date DESC;
```

9.8 Key formulas

Key Formulas

- **B$^+$-tree**: search in $O(\log_m n)$ disk accesses, range scan in $O(\log_m n + k/B)$
- **Hash**: equality lookup in $O(1)$, no range queries
- **Bitmap**: logical operations in $O(n/w)$, ideal for low cardinality
- **R-tree**: spatial search in $O(\log n)$ on average
- **Trade-off**: read acceleration vs. write overhead

9.9 Exercises

Exercise 9.1. Draw the B$^+$-tree of order 4 obtained after sequentially inserting keys $3, 7, 1, 14, 8, 5, 11, 17, 13, 6, 23, 12, 20, 26, 4, 16$. Show the state after deleting key 14.

Exercise 9.2. A table `employees` has 10 million rows. The column `department` has 50 distinct values; the column `salary` is continuous. What type of index do you recommend for each? Justify.

Exercise 9.3. Write a SQL query that *cannot* use a hash index but *can* use a B$^+$-tree index. Explain why.

Exercise 9.4. We have a bitmap index on the column `status` (3 values: active, inactive, suspended) and a bitmap index on `region` (10 values). How many bitwise AND operations are needed to answer `WHERE status = 'active' AND region IN ('NY', 'CA')`?

Exercise 9.5. Use `EXPLAIN ANALYZE` in PostgreSQL to compare the performance of a query with and without an index. Measure the number of pages read (buffers) and execution time.

Chapter 10

ETL and Data Warehousing

Central idea

A data warehouse centralizes enterprise data in a schema optimized for analytical queries. ETL (Extract, Transform, Load) pipelines feed this warehouse by cleaning and transforming data from heterogeneous sources.

10.1 Fundamental concepts

Definition 10.1 (Data warehouse)**.** A **data warehouse** is a database that is *subject-oriented*, *integrated*, *non-volatile*, and *time-variant*, designed for decision support (Inmon, 1992).

Remark 10.2. A data warehouse contrasts with **OLTP** (*Online Transaction Processing*) databases that serve daily operations. Analytical workloads fall under **OLAP** (*Online Analytical Processing*).

Characteristic	OLTP	OLAP
Operations	Short reads/writes	Long analytical queries
Users	Many, operational	Few, analysts
Data	Current, detailed	Historical, aggregated
Schema	Normalized (3NF)	Denormalized (star, snowflake)

10.2 Dimensional modeling

Definition 10.3 (Star schema)**.** A **star schema** organizes data around a central **fact table** (quantitative measures) surrounded by **dimension tables** (descriptive attributes). Dimensions are linked to the fact table via foreign keys.

Example 10.4. Data warehouse for retail sales:

```
-- Fact table
CREATE TABLE fact_sales (
    sale_id      SERIAL PRIMARY KEY,
    product_id   INT REFERENCES dim_product(id),
    customer_id  INT REFERENCES dim_customer(id),
```

```
    date_id      INT REFERENCES dim_date(id),
    store_id     INT REFERENCES dim_store(id),
    quantity     INT,
    amount       DECIMAL(10,2),
    cost         DECIMAL(10,2)
);

-- Dimension table
CREATE TABLE dim_date (
    id         INT PRIMARY KEY,
    full_date DATE,
    month      INT,
    quarter    INT,
    year       INT,
    day_of_week VARCHAR(10)
);
```

Definition 10.5 (Snowflake schema)**.** The **snowflake schema** is a variant where dimension tables are normalized: each hierarchy is extracted into a separate table (e.g., `dim_city` → `dim_region` → `dim_country`).

Attention

The star schema is generally preferred over the snowflake because:

- joins are simpler (single step),
- performance is better (fewer joins),
- redundancy is acceptable in OLAP (disk space is cheap).

10.3 The ETL process

Definition 10.6 (ETL)**.** The **ETL** (*Extract, Transform, Load*) process consists of:

1. **Extract**: retrieve data from sources (OLTP databases, CSV files, APIs, logs).
2. **Transform**: clean, normalize, deduplicate, aggregate, compute derived metrics.
3. **Load**: insert the transformed data into the warehouse.

Example 10.7. ETL pipeline in Python with pandas:

```
import pandas as pd
from sqlalchemy import create_engine

# 1. Extract
df_sales = pd.read_csv("sales_2025.csv")
df_customers = pd.read_sql("SELECT * FROM customers", engine_oltp)

# 2. Transform
```

```python
df_sales["date"] = pd.to_datetime(df_sales["date"])
df_sales = df_sales.dropna(subset=["amount"])
df_sales["amount"] = df_sales["amount"].clip(lower=0)
df = df_sales.merge(df_customers, on="customer_id", how="left")
df["quarter"] = df["date"].dt.quarter

# 3. Load
engine_dw = create_engine("postgresql://user:pass@host/warehouse")
df.to_sql("fact_sales", engine_dw, if_exists="append", index=False)
```

Definition 10.8 (ELT)**.** The **ELT** (*Extract, Load, Transform*) variant loads raw data into the warehouse first, then performs transformations directly in SQL. This approach is popular with cloud warehouses (BigQuery, Snowflake, Redshift) thanks to their compute power.

10.4 Slowly Changing Dimensions (SCD)

Definition 10.9 (Slowly Changing Dimensions)**. SCDs** manage modifications to dimension attributes:

- **Type 1**: overwrite the old value (no history).
- **Type 2**: create a new row with validity dates (`start_date`, `end_date`, `is_current`).
- **Type 3**: add a column for the old value (limited history).

Example 10.10. SCD Type 2 — a customer moves:

```sql
-- Before: current row
-- id=1, name='Smith', city='New York', start_date='2020-01-01',
--        end_date='9999-12-31', is_current=TRUE

-- After relocation:
UPDATE dim_customer SET end_date = '2025-06-15', is_current = FALSE
WHERE id = 1 AND is_current = TRUE;

INSERT INTO dim_customer (natural_id, name, city, start_date, end_date,
↪  is_current)
VALUES (1, 'Smith', 'Boston', '2025-06-15', '9999-12-31', TRUE);
```

10.5 OLAP operations

Definition 10.11 (OLAP operations)**.** The classic OLAP operations on a multidimensional cube are:

- **Roll-up** (aggregation): move up in the hierarchy (day → month → year).
- **Drill-down** (disaggregation): move down in the hierarchy.
- **Slice**: fix one dimension (e.g., `year = 2025`).

- **Dice**: select a multi-dimensional sub-cube.
- **Pivot**: rotate the axes of the cube.

```
-- Roll-up: sales by month -> sales by quarter
SELECT d.quarter, d.year, SUM(f.amount) AS total
FROM fact_sales f
JOIN dim_date d ON f.date_id = d.id
GROUP BY d.quarter, d.year
ORDER BY d.year, d.quarter;

-- Slice + Dice with CUBE
SELECT d.year, p.category, s.region,
       SUM(f.amount) AS total
FROM fact_sales f
JOIN dim_date d ON f.date_id = d.id
JOIN dim_product p ON f.product_id = p.id
JOIN dim_store s ON f.store_id = s.id
WHERE d.year IN (2024, 2025)
GROUP BY CUBE (d.year, p.category, s.region);
```

10.6 Materialized views

Definition 10.12 (Materialized view)**.** A **materialized view** is a view whose result is physically stored on disk. It accelerates repetitive analytical queries at the cost of additional storage and a refresh overhead.

```
-- Creating a materialized view
CREATE MATERIALIZED VIEW mv_monthly_sales AS
SELECT d.year, d.month, p.category,
       SUM(f.amount) AS total, COUNT(*) AS num_sales
FROM fact_sales f
JOIN dim_date d ON f.date_id = d.id
JOIN dim_product p ON f.product_id = p.id
GROUP BY d.year, d.month, p.category;

-- Refreshing
REFRESH MATERIALIZED VIEW CONCURRENTLY mv_monthly_sales;
```

10.7 Key formulas

Key Formulas

- **ETL**: Extract → Transform → Load (vs. ELT: Extract → Load → Transform)
- **Star schema**: 1 fact table + k dimension tables, simple joins
- **OLAP**: Roll-up, Drill-down, Slice, Dice, Pivot

- **SCD Type 2**: historization via `start_date`/`end_date`
- **Materialized view**: space vs. query time trade-off

10.8 Exercises

Exercise 10.1. Design a star schema for an airline reservation system. Identify the fact table, measures, and at least four dimensions.

Exercise 10.2. Transform the star schema from the previous exercise into a snowflake schema. Discuss the advantages and disadvantages.

Exercise 10.3. Write an ETL pipeline in Python that: (a) extracts data from a JSON file, (b) cleans missing values and normalizes dates, (c) loads the result into a PostgreSQL table.

Exercise 10.4. Implement a Type 2 slowly changing dimension for the `dim_product` table. Simulate a price change and verify that history is preserved.

Exercise 10.5. Write the SQL queries corresponding to the following OLAP operations on the sales schema: (a) roll-up from day to quarter, (b) slice on the year 2025, (c) dice on electronics products sold in California.

Chapter 11

NoSQL Databases

Central idea

NoSQL (*Not Only SQL*) databases abandon the strict relational model to gain horizontal scalability, schema flexibility, and performance on specific use cases. Four major families exist: key-value, document, column-family, and graph.

11.1 Motivation and context

Relational databases reach their limits when facing:

- massive volumes (petabytes),
- semi-structured or unstructured data,
- high availability and geographic distribution requirements,
- evolving schemas (agile development).

Remark 11.1. NoSQL does not mean "anti-SQL" but "Not Only SQL." Many NoSQL databases now offer a declarative query language (CQL for Cassandra, MQL for MongoDB, Cypher for Neo4j).

11.2 The CAP theorem

Theorem 11.2 (CAP theorem (Brewer, 2000))**.** *A distributed system can simultaneously guarantee at most two of the following three properties:*

1. ***Consistency*** *— every read returns the most recent write.*
2. ***Availability*** *— every request receives a response (not necessarily the most recent data).*
3. ***Partition tolerance*** *— the system continues to operate despite network partitions.*

Attention

In practice, network partitions are unavoidable in a distributed system. The real choice is therefore between **CP** (strong consistency, e.g., HBase, MongoDB with `readConcern: majority`) and **AP** (availability, e.g., Cassandra, DynamoDB in eventual mode).

Definition 11.3 (Eventual consistency)**.** A system with **eventual consistency** guarantees that, in the absence of new writes, all replicas converge to the same value after a finite delay.

11.3 Key-value stores

Definition 11.4 (Key-value store)**.** A **key-value store** stores pairs (k, v) where k is a unique key and v is an opaque value (string, blob, serialized object). Access is by key only: `GET`, `SET`, `DELETE`.

Example 11.5. Redis: in-memory cache and data structure server:

```
import redis

r = redis.Redis(host='localhost', port=6379, db=0)
r.set('user:1001:name', 'Alice')
r.set('user:1001:email', 'alice@example.com')
r.expire('user:1001:name', 3600)  # 1-hour TTL

name = r.get('user:1001:name')  # b'Alice'

# Advanced data structures
r.hset('user:1002', mapping={'name': 'Bob', 'age': '25'})
r.lpush('queue:jobs', 'task_42')
r.zadd('leaderboard', {'Alice': 1500, 'Bob': 1200})
```

Remark 11.6. Typical use cases: application cache, user sessions, real-time counters, message queues.

11.4 Document stores

Definition 11.7 (Document store)**.** A **document store** stores semi-structured documents (JSON, BSON) grouped into *collections*. Each document has a flexible schema: two documents in the same collection may have different fields.

Example 11.8. MongoDB:

```
from pymongo import MongoClient

client = MongoClient('mongodb://localhost:27017/')
db = client['university']
```

```
col = db['students']

# Insertion
col.insert_one({
    'last_name': 'Smith',
    'first_name': 'Alice',
    'grades': [
        {'course': 'Databases', 'grade': 92},
        {'course': 'Algorithms', 'grade': 88}
    ],
    'address': {'city': 'New York', 'zip': '10001'}
})

# Query with filter and projection
results = col.find(
    {'grades.grade': {'$gte': 90}},
    {'last_name': 1, 'first_name': 1, '_id': 0}
)

# Aggregation (equivalent to GROUP BY)
pipeline = [
    {'$unwind': '$grades'},
    {'$group': {'_id': '$last_name', 'avg_grade': {'$avg': '$grades.grade'}}}
]
col.aggregate(pipeline)
```

Remark 11.9. MongoDB offers secondary indexes, multi-document transactions (since version 4.0), and a rich aggregation language. It suits data with variable schemas (product catalogs, CMS, IoT).

11.5 Column-family stores

Definition 11.10 (Column-family store)**.** A **column-family store** organizes data by column families rather than rows. Each row is identified by a partition key and can have a variable number of columns.

Example 11.11. Apache Cassandra:

```
-- CQL (Cassandra Query Language)
CREATE KEYSPACE university WITH replication = {
    'class': 'SimpleStrategy', 'replication_factor': 3
};

CREATE TABLE university.grades (
    student_id UUID,
    course TEXT,
    semester INT,
    grade FLOAT,
    PRIMARY KEY ((student_id), course, semester)
) WITH CLUSTERING ORDER BY (course ASC, semester DESC);
```

```
-- The partition key (student_id) determines the storage node
-- Clustering columns order data within the partition
```

Remark 11.12. Cassandra excels at massive distributed writes (logs, time series, IoT). Modeling is query-driven: tables are designed based on the `SELECT` queries you plan to execute.

11.6 Graph databases

Definition 11.13 (Graph database)**.** A **graph database** stores **nodes** (entities) and **edges** (relationships) as first-class citizens. Graph traversals are native operations running in $O(1)$ per relationship (independent of graph size).

Example 11.14. Neo4j and the Cypher language:

```
// Create nodes and relationships
CREATE (a:Person {name: 'Alice', age: 30})
CREATE (b:Person {name: 'Bob', age: 25})
CREATE (a)-[:FRIENDS_WITH {since: 2020}]->(b)
CREATE (a)-[:WORKS_AT]->(:Company {name: 'TechCorp'})

// Find friends of friends (distance 2)
MATCH (p:Person {name: 'Alice'})-[:FRIENDS_WITH*2]-(fof)
WHERE fof <> p
RETURN DISTINCT fof.name

// Shortest path
MATCH path = shortestPath(
    (a:Person {name: 'Alice'})-[*..6]-(b:Person {name: 'Charlie'})
)
RETURN path
```

Remark 11.15. Graph databases are ideal for social networks, fraud detection, recommendation systems, and knowledge graphs. They outperform relational databases when queries involve deep recursive joins.

11.7 Comparison and selection criteria

	Key-value	**Document**	**Column**	**Graph**
Example	Redis	MongoDB	Cassandra	Neo4j
Schema	None	Flexible	Semi-fixed	Flexible
Queries	By key	Rich	By partition	Traversals
Scalability	Very high	High	Very high	Moderate
Typical use	Cache	CMS, catalog	Time series	Social network

Proposition 11.16 (Selection criteria)**.** The choice of NoSQL database type depends on:

1. the **structure** of the data (fixed vs. flexible schema),

2. **access patterns** (by key, complex queries, traversals),
3. **consistency** vs. **availability** requirements,
4. the **volume** and **velocity** of the data.

11.8 Key formulas

Key Formulas

- **CAP**: Consistency + Availability + Partition tolerance — at most 2 out of 3
- **Key-value**: $O(1)$ access by key, no queries on values
- **Document**: flexible schema, secondary indexes, aggregation
- **Column-family**: query-driven modeling, massive distributed writes
- **Graph**: traversals in $O(k)$ where k = number of edges traversed

11.9 Exercises

Exercise 11.1. For each of the following use cases, recommend a NoSQL database type and justify: (a) web session cache, (b) product catalog with variable attributes, (c) IoT sensor tracking at 100,000 events/s, (d) community detection in a social network.

Exercise 11.2. Write a Python script that inserts 10,000 documents into MongoDB, creates an index on a field, then compares the query time with and without the index.

Exercise 11.3. Model a movie recommendation system in Neo4j. Create nodes (users, movies, genres) and relationships (WATCHED, RATED, HAS_GENRE). Write a Cypher query to recommend movies to a user based on their friends' preferences.

Exercise 11.4. Design a Cassandra schema for a messaging application (messages between users). The primary query is "display the last N messages in a conversation." Justify the choice of partition key and clustering columns.

Exercise 11.5. Discuss the implications of the CAP theorem for a banking application distributed across multiple continents. What trade-off would you recommend? Why?

Bibliography

[1] Ramakrishnan, R. and Gehrke, J., *Database Management Systems*, 3rd ed., McGraw-Hill, 2003.

[2] Date, C.J., *An Introduction to Database Systems*, 8th ed., Pearson, 2004.

[3] Silberschatz, A., Korth, H.F. and Sudarshan, S., *Database System Concepts*, 7th ed., McGraw-Hill, 2020.