CSL303/MAL505 — Database Management Systems

Module I: Relational Foundations & SQL
Lecture 1: Why Databases?

Amit Kumar Dhar

July 28,2026

IIT Bhilai

Today’s agenda

- **Who we are & how this course runs** — logistics, grading, tools

Today’s agenda

- **Who we are & how this course runs** — logistics, grading, tools
- **Motivation** — why databases are everywhere and worth learning

Today’s agenda

- **Who we are & how this course runs** — logistics, grading, tools
- **Motivation** — why databases are everywhere and worth learning
- **The core problem** — what is data, a database, a DBMS?

Today's agenda

- **Who we are & how this course runs** — logistics, grading, tools
- **Motivation** — why databases are everywhere and worth learning
- **The core problem** — what is data, a database, a DBMS?
- **The old way** — why not just use files?

Today's agenda

- **Who we are & how this course runs** — logistics, grading, tools
- **Motivation** — why databases are everywhere and worth learning
- **The core problem** — what is data, a database, a DBMS?
- **The old way** — why not just use files?
- **The DBMS solution** — the services a DBMS provides

Today's agenda

- **Who we are & how this course runs** — logistics, grading, tools
- **Motivation** — why databases are everywhere and worth learning
- **The core problem** — what is data, a database, a DBMS?
- **The old way** — why not just use files?
- **The DBMS solution** — the services a DBMS provides
- **Architecture** — the three-schema view & data independence

Today's agenda

- **Who we are & how this course runs** — logistics, grading, tools
- **Motivation** — why databases are everywhere and worth learning
- **The core problem** — what is data, a database, a DBMS?
- **The old way** — why not just use files?
- **The DBMS solution** — the services a DBMS provides
- **Architecture** — the three-schema view & data independence
- **What's inside** — a peek at the components we'll build this term

Course Logistics

About this course

- **Goal:** learn to *use* databases well **and** understand how they *work inside*.

About this course

- **Goal:** learn to *use* databases well **and** understand how they *work inside*.
- Two halves that reinforce each other:

About this course

- **Goal:** learn to *use* databases well **and** understand how they *work inside*.
- Two halves that reinforce each other:
  - **Use it:** deep SQL, on real engines (SQLite + PostgreSQL).

About this course

- **Goal:** learn to *use* databases well **and** understand how they *work inside*.
- Two halves that reinforce each other:
  - **Use it:** deep SQL, on real engines (SQLite + PostgreSQL).
  - **Build it:** implement real DBMS components in Python.

About this course

- **Goal:** learn to *use* databases well **and** understand how they *work inside*.
- Two halves that reinforce each other:
  - **Use it:** deep SQL, on real engines (SQLite + PostgreSQL).
  - **Build it:** implement real DBMS components in Python.
- By the end you will have **written parts of a database engine** yourself:

About this course

- **Goal:** learn to *use* databases well **and** understand how they *work inside*.
- Two halves that reinforce each other:
  - **Use it:** deep SQL, on real engines (SQLite + PostgreSQL).
  - **Build it:** implement real DBMS components in Python.
- By the end you will have **written parts of a database engine** yourself:

  A page store, a B+tree, a buffer pool, and a write-ahead log.

Course format

- **Lectures:** 3 per week, 1 hour each. Attend — concepts build on each other.

Course format

- **Lectures:** 3 per week, 1 hour each. Attend — concepts build on each other.
- **Labs:** 1 per week, 2 hours each. This is where the learning sticks.

Course format

- **Lectures:** 3 per week, 1 hour each. Attend — concepts build on each other.
- **Labs:** 1 per week, 2 hours each. This is where the learning sticks.
- **Quizzes:** announced *and* surprise (keeps you current).

Course format

- **Lectures:** 3 per week, 1 hour each. Attend — concepts build on each other.
- **Labs:** 1 per week, 2 hours each. This is where the learning sticks.
- **Quizzes:** announced *and* surprise (keeps you current).
- **Lab evaluations:** some announced, some on the spot.

Course format

- **Lectures:** 3 per week, 1 hour each. Attend — concepts build on each other.
- **Labs:** 1 per week, 2 hours each. This is where the learning sticks.
- **Quizzes:** announced *and* surprise (keeps you current).
- **Lab evaluations:** some announced, some on the spot.
- **Exams:** Mid-Sem and End-Sem.

Course format

- **Lectures:** 3 per week, 1 hour each. Attend — concepts build on each other.
- **Labs:** 1 per week, 2 hours each. This is where the learning sticks.
- **Quizzes:** announced *and* surprise (keeps you current).
- **Lab evaluations:** some announced, some on the spot.
- **Exams:** Mid-Sem and End-Sem.
- Possible **project/Assignment** in the final weeks.

Course format

- **Lectures:** 3 per week, 1 hour each. Attend — concepts build on each other.
- **Labs:** 1 per week, 2 hours each. This is where the learning sticks.
- **Quizzes:** announced *and* surprise (keeps you current).
- **Lab evaluations:** some announced, some on the spot.
- **Exams:** Mid-Sem and End-Sem.
- Possible **project/Assignment** in the final weeks.

(Course website to be notified later.)

Suggested grading

Component	Weight
Mid-Sem exam	20%
End-Sem exam	30%
Lab work (incl. 4 build labs)	30%
Quizzes (may be Projects/Assignments)	15%
Attendance	5%

Suggested grading

Component	Weight
Mid-Sem exam	20%
End-Sem exam	30%
Lab work (incl. 4 build labs)	30%
Quizzes (may be Projects/Assignments)	15%
Attendance	5%

The **labs are 30%** — this is a hands-on course. Do them yourself.

Books

- **Silberschatz, Korth, Sudarshan** — *Database System Concepts* (primary)
- **Garcia-Molina, Ullman, Widom** — *Database Systems: The Complete Book*
- **Elmasri, Navathe** — *Fundamentals of Database Systems*
- **Petrov** — *Database Internals* (for the internals module)
- **Sibsankar Haldar** — *Inside SQLite* (storage internals)

Tools you'll install

- **SQLite** — zero-setup, a database is a single file. Weeks 1–4, 7.

Tools you'll install

- **SQLite** — zero-setup, a database is a single file. Weeks 1–4, 7.
- **DB Browser for SQLite** — a friendly GUI.

Tools you'll install

- **SQLite** — zero-setup, a database is a single file. Weeks 1–4, 7.
- **DB Browser for SQLite** — a friendly GUI.
- **PostgreSQL** — a "real" server DBMS. From Week 4 on.

Tools you'll install

- **SQLite** — zero-setup, a database is a single file. Weeks 1–4, 7.
- **DB Browser for SQLite** — a friendly GUI.
- **PostgreSQL** — a "real" server DBMS. From Week 4 on.
- **Python 3.10+** — to script queries and to *build* engine components.

Tools you'll install

- **SQLite** — zero-setup, a database is a single file. Weeks 1–4, 7.
- **DB Browser for SQLite** — a friendly GUI.
- **PostgreSQL** — a "real" server DBMS. From Week 4 on.
- **Python 3.10+** — to script queries and to *build* engine components.

Setup guide: Install them. **Do this before Lab 1.**

Part 1 — Motivation

The data explosion

- Every company is now a **data company**.

The data explosion

- Every company is now a **data company**.
- Social media, streaming, e-commerce, IoT, sensors, logs...

The data explosion

- Every company is now a **data company**.
- Social media, streaming, e-commerce, IoT, sensors, logs...
- Data volumes double every couple of years.

The data explosion

- Every company is now a **data company**.
- Social media, streaming, e-commerce, IoT, sensors, logs…
- Data volumes double every couple of years.
- The value isn’t the raw bytes — it’s the ability to **store, protect, and answer questions about** them reliably.

The data explosion

- Every company is now a **data company**.
- Social media, streaming, e-commerce, IoT, sensors, logs…
- Data volumes double every couple of years.
- The value isn't the raw bytes — it's the ability to **store, protect, and answer questions about** them reliably.
- Databases are the technology that makes that possible.

Databases are a career

- **Backend Developer** — nearly every app has a database behind it.

Databases are a career

- **Backend Developer** — nearly every app has a database behind it.
- **Data Engineer** — builds pipelines and infrastructure at scale.

Databases are a career

- **Backend Developer** — nearly every app has a database behind it.
- **Data Engineer** — builds pipelines and infrastructure at scale.
- **Database Administrator (DBA)** — performance, security, reliability.

Databases are a career

- **Backend Developer** — nearly every app has a database behind it.
- **Data Engineer** — builds pipelines and infrastructure at scale.
- **Database Administrator (DBA)** — performance, security, reliability.
- **Data Analyst / Scientist** — queries data for insight and models.

Databases are a career

- **Backend Developer** — nearly every app has a database behind it.
- **Data Engineer** — builds pipelines and infrastructure at scale.
- **Database Administrator (DBA)** — performance, security, reliability.
- **Data Analyst / Scientist** — queries data for insight and models.

These are among the most in-demand, well-paid roles in tech — and they all rest on the concepts in this course.

Databases are active research

- **Distributed & cloud databases** — spanning thousands of machines (Google Spanner, Amazon Aurora, CockroachDB).

Databases are active research

- **Distributed & cloud databases** — spanning thousands of machines (Google Spanner, Amazon Aurora, CockroachDB).
- **ML for databases** — can a DB tune and index itself?

Databases are active research

- **Distributed & cloud databases** — spanning thousands of machines (Google Spanner, Amazon Aurora, CockroachDB).
- **ML for databases** — can a DB tune and index itself?
- **New data models** — graph, time-series, vector databases for AI.

Databases are active research

- **Distributed & cloud databases** — spanning thousands of machines (Google Spanner, Amazon Aurora, CockroachDB).
- **ML for databases** — can a DB tune and index itself?
- **New data models** — graph, time-series, vector databases for AI.
- **Hardware-conscious engines** — GPUs, NVMe, persistent memory.

Databases are active research

- **Distributed & cloud databases** — spanning thousands of machines (Google Spanner, Amazon Aurora, CockroachDB).
- **ML for databases** — can a DB tune and index itself?
- **New data models** — graph, time-series, vector databases for AI.
- **Hardware-conscious engines** — GPUs, NVMe, persistent memory.

A DBMS is one of the most sophisticated pieces of software ever built.

Part 2 — Core concepts

What is data?

- **Data** = raw, unorganized facts.

What is data?

- **Data** = raw, unorganized facts.
- Examples: a customer's name, a product's price, a transaction's timestamp.

What is data?

- **Data** = raw, unorganized facts.
- Examples: a customer's name, a product's price, a transaction's timestamp.
- An individual fact is a **data item**.

What is data?

- **Data** = raw, unorganized facts.
- Examples: a customer's name, a product's price, a transaction's timestamp.
- An individual fact is a **data item**.
- Data by itself has little value — it becomes useful when it is **organized, related, and queryable**.

What is a database?

- A **database** is a collection of **related** data.

What is a database?

- A **database** is a collection of **related** data.
- It models some slice of the real world — a "**miniworld**."

What is a database?

- A **database** is a collection of **related** data.
- It models some slice of the real world — a "**miniworld**."
- It is **logically coherent** with inherent meaning (not a random pile of bytes).

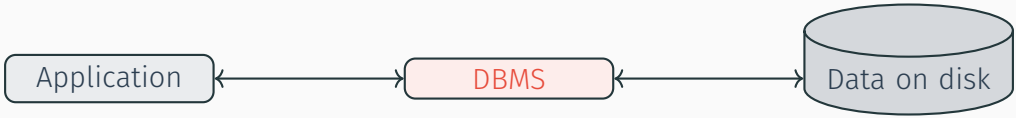
What is a database?

- A **database** is a collection of **related** data.
- It models some slice of the real world — a "**miniworld**."
- It is **logically coherent** with inherent meaning (not a random pile of bytes).

Example:
A *University* database holds related data about students, instructors, courses, sections, and grades. We'll use exactly this University database all term.

What is a DBMS?

- A **Database Management System** is the software that lets us **define, store, manage, and query** a database.
- It sits as a **layer between the user/application and the physical data**.
- Examples: **PostgreSQL, MySQL, SQLite, Oracle, SQL Server, MongoDB.**



Database vs DBMS — don’t confuse them

- **Database** = the data (the University’s students, courses, grades).

Database vs DBMS — don’t confuse them

- **Database** = the data (the University’s students, courses, grades).
- **DBMS** = the software (PostgreSQL) that manages *many* databases.

Database vs DBMS — don’t confuse them

- **Database** = the data (the University’s students, courses, grades).
- **DBMS** = the software (PostgreSQL) that manages *many* databases.
- One DBMS process can host many independent databases.

Database vs DBMS — don’t confuse them

- **Database** = the data (the University’s students, courses, grades).
- **DBMS** = the software (PostgreSQL) that manages *many* databases.
- One DBMS process can host many independent databases.
- ”Database” is often used loosely to mean both — but the distinction matters.

Real-world example: Amazon

A single order touches a lot of data the DBMS manages:

- **Products** — names, descriptions, prices, inventory.
- **Customers** — logins, addresses, payment methods.
- **Orders** — items, quantities, shipping status.
- **Reviews** — ratings, comments.

Millions of users, all at once, expecting it to be **fast, correct, and safe**.

Part 3 — The old way

The file-based approach

- You *could* store everything in ordinary files: CSV, text, binary.

The file-based approach

- You *could* store everything in ordinary files: CSV, text, binary.
- Each application reads/writes its own files, in its own format.

The file-based approach

- You *could* store everything in ordinary files: CSV, text, binary.
- Each application reads/writes its own files, in its own format.
- This is how things worked **before** DBMSs — and it breaks down fast.

The file-based approach

- You *could* store everything in ordinary files: CSV, text, binary.
- Each application reads/writes its own files, in its own format.
- This is how things worked **before** DBMSs — and it breaks down fast.
- Let's use a simple **bank** application to see why.

Problem 1: Redundancy & inconsistency

- The same fact gets duplicated across files.

Problem 1: Redundancy & inconsistency

- The same fact gets duplicated across files.
- A customer's address in both `checking.csv` and `savings.csv`.

Problem 1: Redundancy & inconsistency

- The same fact gets duplicated across files.
- A customer's address in both `checking.csv` and `savings.csv`.
- Customer moves $\rightarrow$ you must update **both**.

Problem 1: Redundancy & inconsistency

- The same fact gets duplicated across files.
- A customer's address in both `checking.csv` and `savings.csv`.
- Customer moves → you must update **both**.
- Update one and forget the other → the data is now **inconsistent**.

Problem 1: Redundancy & inconsistency

- The same fact gets duplicated across files.
- A customer's address in both `checking.csv` and `savings.csv`.
- Customer moves $\rightarrow$ you must update **both**.
- Update one and forget the other $\rightarrow$ the data is now **inconsistent**.
- Consequences: bills to the wrong address, missed fraud alerts, bad decisions.

Problem 2: Difficulty accessing data

- Every new question needs a **new program**.

Problem 2: Difficulty accessing data

- Every new question needs a **new program**.
- *"Total balance of customers in California with more than two accounts?"*

Problem 2: Difficulty accessing data

- Every new question needs a **new program**.
- *"Total balance of customers in California with more than two accounts?"*
- With files you must hand-write code to:

Problem 2: Difficulty accessing data

- Every new question needs a **new program**.
- *"Total balance of customers in California with more than two accounts?"*
- With files you must hand-write code to:

  parse each file → join by customer → filter by state → count accounts → sum.

Problem 2: Difficulty accessing data

- Every new question needs a **new program**.
- *"Total balance of customers in California with more than two accounts?"*
- With files you must hand-write code to:

  parse each file → join by customer → filter by state → count accounts → sum.
- Slow to write, easy to get wrong, thrown away after one use.

Problem 3: Data isolation

- Data scattered across many files, in many formats (CSV, binary, JSON...).
- Combining them requires custom glue code every time.

Problem 3 & 4

Problem 3: Data isolation

- Data scattered across many files, in many formats (CSV, binary, JSON…).
- Combining them requires custom glue code every time.

Problem 4: Integrity problems

- Business rules are hard to enforce in files:
  - a balance must never be negative;
  - account type must be *Checking*, *Savings*, or *Loan*;
- In a file system, **every application** must re-implement these checks.

Problem 5 & 6

Problem 5: Atomicity failures

- Transfer ₹100 from A to B: 1. debit ₹100 from A, 2. credit ₹100 to B.
- Crash **after step 1, before step 2** $\rightarrow$ ₹100 vanishes.
- Operation must be **all-or-nothing (atomic)**. Files don't do this.

Problem 5 & 6

Problem 5: Atomicity failures

- Transfer ₹100 from A to B: 1. debit ₹100 from A, 2. credit ₹100 to B.
- Crash **after step 1, before step 2** $\rightarrow$ ₹100 vanishes.
- Operation must be **all-or-nothing (atomic)**. Files don't do this.

Problem 6: Concurrent-access anomalies

- Teller 1 & 2 read ₹500 at the same time.
- Teller 1 deposits ₹100 $\rightarrow$ writes ₹600.
- Teller 2 withdraws ₹50 $\rightarrow$ writes ₹450.
- Final balance ₹450 — **Teller 1's deposit was lost** (Lost update).

Problem 7: Security problems

- Different users need different access:

Problem 7: Security problems

- Different users need different access:
  - a teller may see balances but not employee salaries;

Problem 7: Security problems

- Different users need different access:
  - a teller may see balances but not employee salaries;
  - a customer may see only *their own* accounts.

Problem 7: Security problems

- Different users need different access:
  - a teller may see balances but not employee salaries;
  - a customer may see only *their own* accounts.
- File-system permissions are too coarse to express this.

Problem 7: Security problems

- Different users need different access:
  - a teller may see balances but not employee salaries;
  - a customer may see only *their own* accounts.
- File-system permissions are too coarse to express this.
- You need **fine-grained, data-level** access control.

Seven problems, one conclusion

1. Redundancy & inconsistency
2. Difficulty accessing data
3. Data isolation
4. Integrity problems
5. Atomicity failures
6. Concurrency anomalies
7. Security problems

A DBMS exists to solve **all seven** — that's its reason to exist.

Part 4 — The DBMS solution

Feature 1 & 2

Feature 1: A data model

- Structures data with a **model** — e.g. the **relational model** (tables).
- Reduces redundancy; makes relationships explicit.

Feature 1 & 2

Feature 1: A data model

- Structures data with a **model** — e.g. the **relational model** (tables).
- Reduces redundancy; makes relationships explicit.

Feature 2: A high-level query language

- Ask **what** you want, not **how** to compute it — *declarative*.

```
SELECT c.name, SUM(a.balance)
FROM   customer c JOIN account a ON c.cust_id = a.cust_id
WHERE  c.state = 'California'
GROUP  BY c.cust_id, c.name HAVING COUNT(*) > 2;
```

Feature 3: Integrity constraints

- Declare rules once; the DBMS enforces them for **every** application.

```
CREATE TABLE Accounts (
    account_no  INTEGER PRIMARY KEY,
    balance     NUMERIC CHECK (balance >= 0),
    acct_type   TEXT    CHECK (acct_type IN ('Checking','Savings','Loan'))
);
```

Feature 3: Integrity constraints

- Declare rules once; the DBMS enforces them for **every** application.

```
CREATE TABLE Accounts (
    account_no  INTEGER PRIMARY KEY,
    balance     NUMERIC CHECK (balance >= 0),
    acct_type   TEXT    CHECK (acct_type IN ('Checking','Savings','Loan'))
);
```

Feature 4: Transactions & ACID

- **A**tomicity, **C**onsistency, **I**solation, **D**urability.
- The fund transfer becomes safe: both steps commit together or neither does.

Feature 5, 6 & 7

Feature 5: Concurrency control

- Uses **locking** and **multi-versioning (MVCC)**.
- The ”lost update” from before simply **cannot happen**.

Feature 5, 6 & 7

Feature 5: Concurrency control

- Uses **locking** and **multi-versioning (MVCC)**.
- The ”lost update” from before simply **cannot happen**.

Feature 6: Recovery

- Uses a **write-ahead log** to restore a consistent state after a crash.

Feature 5, 6 & 7

Feature 5: Concurrency control

- Uses **locking** and **multi-versioning (MVCC)**.
- The ”lost update” from before simply **cannot happen**.

Feature 6: Recovery

- Uses a **write-ahead log** to restore a consistent state after a crash.

Feature 7: Security & administration

- Users, roles, and privileges: `GRANT SELECT ON ...`

Amazon, revisited

- **Integrity** — prices are positive, quantities are valid.

Amazon, revisited

- **Integrity** — prices are positive, quantities are valid.
- **Transactions** — payment + inventory + order created atomically.

Amazon, revisited

- **Integrity** — prices are positive, quantities are valid.
- **Transactions** — payment + inventory + order created atomically.
- **Concurrency** — millions shop simultaneously without interference.

Amazon, revisited

- **Integrity** — prices are positive, quantities are valid.
- **Transactions** — payment + inventory + order created atomically.
- **Concurrency** — millions shop simultaneously without interference.
- **Recovery** — a datacenter hiccup doesn't lose your order.

Amazon, revisited

- **Integrity** — prices are positive, quantities are valid.
- **Transactions** — payment + inventory + order created atomically.
- **Concurrency** — millions shop simultaneously without interference.
- **Recovery** — a datacenter hiccup doesn't lose your order.
- **Security** — you see only your own history.

Amazon, revisited

- **Integrity** — prices are positive, quantities are valid.
- **Transactions** — payment + inventory + order created atomically.
- **Concurrency** — millions shop simultaneously without interference.
- **Recovery** — a datacenter hiccup doesn't lose your order.
- **Security** — you see only your own history.

Every DBMS feature maps to a real requirement.

Part 5 — Architecture

Data abstraction: three levels

- **Physical level** — *how* data is actually stored (files, pages, bytes).

Data abstraction: three levels

- **Physical level** — *how* data is actually stored (files, pages, bytes).
- **Logical level** — *what* data is stored and the relationships (tables, columns).

Data abstraction: three levels

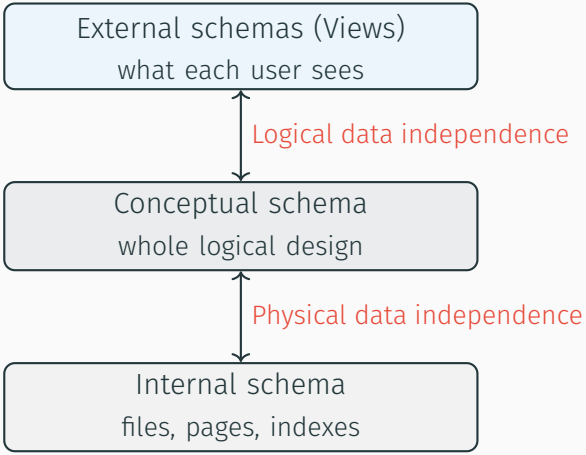
- **Physical level** — *how* data is actually stored (files, pages, bytes).
- **Logical level** — *what* data is stored and the relationships (tables, columns).
- **View level** — tailored slices for particular users/apps.

Data abstraction: three levels

- **Physical level** — *how* data is actually stored (files, pages, bytes).
- **Logical level** — *what* data is stored and the relationships (tables, columns).
- **View level** — tailored slices for particular users/apps.

Users work at the logical/view level; the DBMS handles the physical level.

The three-schema architecture & Data independence



- **Data independence** makes systems maintainable over decades.

Schema vs instance

- **Schema** — the *design*: the structure of the database (rarely changes).

Schema vs instance

- **Schema** — the *design*: the structure of the database (rarely changes).
  - e.g., ”students have an id, name, department, total credits.”

Schema vs instance

- **Schema** — the *design*: the structure of the database (rarely changes).
  - e.g., ”students have an id, name, department, total credits.”
- **Instance** — the *data* at a moment in time (changes constantly).

Schema vs instance

- **Schema** — the *design*: the structure of the database (rarely changes).
  - e.g., "students have an id, name, department, total credits."
- **Instance** — the *data* at a moment in time (changes constantly).
  - e.g., "student 12345 is Shankar, Comp. Sci., 32 credits."

Schema vs instance

- **Schema** — the *design*: the structure of the database (rarely changes).
  - e.g., ”students have an id, name, department, total credits.”
- **Instance** — the *data* at a moment in time (changes constantly).
  - e.g., ”student 12345 is Shankar, Comp. Sci., 32 credits.”

Analogy: schema is a ***type/class****; instance is a* ***value/object****.*

Languages & Users of a DBMS

Languages

- **DDL**: Define schema (`CREATE`)
- **DML**: Query/change data (`SELECT`, `INSERT`)
- **DCL**: Privileges (`GRANT`)
- **TCL**: Transactions (`COMMIT`)

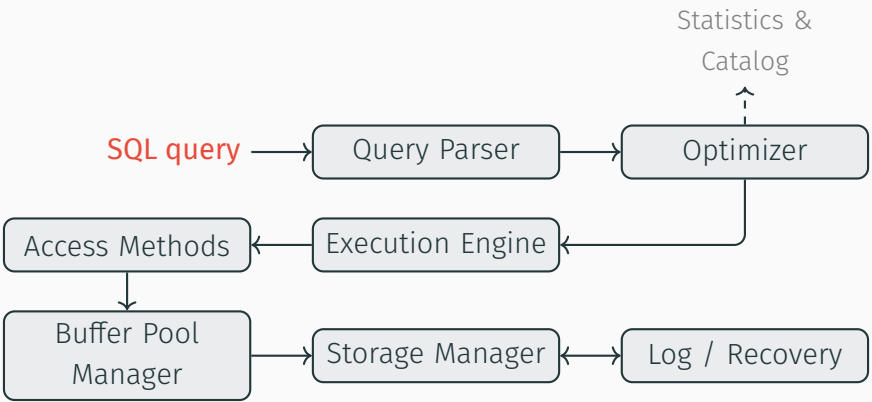
SQL bundles all of these.

Users

- **Naive users**: via apps
- **App programmers**: embed SQL
- **Analysts**: ad-hoc SQL
- **DBAs**: schema, tuning, security

Part 6 — Under the hood

Inside a DBMS (bird's-eye view)



What we'll build this term (in Python)

- **Week 7 — Heap file / slotted pages:** the storage manager's core.

What we'll build this term (in Python)

- **Week 7 — Heap file / slotted pages:** the storage manager's core.
- **Week 8 — B+tree:** the workhorse index (access methods).

What we'll build this term (in Python)

- **Week 7 — Heap file / slotted pages:** the storage manager's core.
- **Week 8 — B+tree:** the workhorse index (access methods).
- **Week 9 — Buffer pool:** caching disk pages in memory with LRU.

What we'll build this term (in Python)

- **Week 7 — Heap file / slotted pages:** the storage manager's core.
- **Week 8 — B+tree:** the workhorse index (access methods).
- **Week 9 — Buffer pool:** caching disk pages in memory with LRU.
- **Week 12 — Write-ahead log:** crash recovery.

What we'll build this term (in Python)

- **Week 7 — Heap file / slotted pages:** the storage manager's core.
- **Week 8 — B+tree:** the workhorse index (access methods).
- **Week 9 — Buffer pool:** caching disk pages in memory with LRU.
- **Week 12 — Write-ahead log:** crash recovery.

You'll see those boxes turn into code you wrote.

Row vs Column Store & OLTP vs OLAP

- **Row store** — store all columns of a row together. Great for **OLTP** (Online Transaction Processing). e.g., Postgres.

Row vs Column Store & OLTP vs OLAP

- **Row store** — store all columns of a row together. Great for **OLTP** (Online Transaction Processing). e.g., Postgres.
- **Column store** — store each column together. Great for **OLAP** (Online Analytical Processing). e.g., DuckDB.

Row vs Column Store & OLTP vs OLAP

- **Row store** — store all columns of a row together. Great for **OLTP** (Online Transaction Processing). e.g., Postgres.
- **Column store** — store each column together. Great for **OLAP** (Online Analytical Processing). e.g., DuckDB.

Row vs Column Store & OLTP vs OLAP

- **Row store** — store all columns of a row together. Great for **OLTP** (Online Transaction Processing). e.g., Postgres.
- **Column store** — store each column together. Great for **OLAP** (Online Analytical Processing). e.g., DuckDB.

- **OLTP**: Many short transactions (e.g. transfer ₹100). Correctness & concurrency matter.

Row vs Column Store & OLTP vs OLAP

- **Row store** — store all columns of a row together. Great for **OLTP** (Online Transaction Processing). e.g., Postgres.
- **Column store** — store each column together. Great for **OLAP** (Online Analytical Processing). e.g., DuckDB.

- **OLTP**: Many short transactions (e.g. transfer ₹100). Correctness & concurrency matter.
- **OLAP**: Few, huge analytical queries (e.g. revenue by region). Throughput matters.

SQL vs NoSQL (preview)

- **Relational/SQL** — tables, schema, strong consistency, joins. Our main focus.

SQL vs NoSQL (preview)

- **Relational/SQL** — tables, schema, strong consistency, joins. Our main focus.
- **NoSQL** — key-value, document, column-family, graph. Trade some guarantees for scale/flexibility. We spend Week 13 here.

SQL vs NoSQL (preview)

- **Relational/SQL** — tables, schema, strong consistency, joins. Our main focus.
- **NoSQL** — key-value, document, column-family, graph. Trade some guarantees for scale/flexibility. We spend Week 13 here.
- Not ”better/worse” — different tools for different jobs.

The map of this course

- **Weeks 1–4:** relational model + SQL (use it well).

The map of this course

- **Weeks 1–4:** relational model + SQL (use it well).
- **Weeks 5–6:** design — ER modelling & normalization.

The map of this course

- **Weeks 1–4:** relational model + SQL (use it well).
- **Weeks 5–6:** design — ER modelling & normalization.
- **Weeks 7–12:** internals — storage, indexing, execution, transactions, recovery (build it).

The map of this course

- **Weeks 1–4:** relational model + SQL (use it well).
- **Weeks 5–6:** design — ER modelling & normalization.
- **Weeks 7–12:** internals — storage, indexing, execution, transactions, recovery (build it).
- **Week 13:** NoSQL & distributed.

The map of this course

- **Weeks 1–4:** relational model + SQL (use it well).
- **Weeks 5–6:** design — ER modelling & normalization.
- **Weeks 7–12:** internals — storage, indexing, execution, transactions, recovery (build it).
- **Week 13:** NoSQL & distributed.
- **Week 14:** advanced topics, wrap-up.

Summary — key takeaways

- A DBMS is **not just a data container**; it's a complex system providing a data model, a query language, integrity, transactions, concurrency, recovery, and security.

Summary — key takeaways

- A DBMS is **not just a data container**; it's a complex system providing a data model, a query language, integrity, transactions, concurrency, recovery, and security.
- File-based systems fail on all seven counts — that's why DBMSs exist.

Summary — key takeaways

- A DBMS is **not just a data container**; it's a complex system providing a data model, a query language, integrity, transactions, concurrency, recovery, and security.
- File-based systems fail on all seven counts — that's why DBMSs exist.
- **Data independence** (via the three-schema architecture) makes systems maintainable.

Summary — key takeaways

- A DBMS is **not just a data container**; it's a complex system providing a data model, a query language, integrity, transactions, concurrency, recovery, and security.
- File-based systems fail on all seven counts — that's why DBMSs exist.
- **Data independence** (via the three-schema architecture) makes systems maintainable.
- We will both **use** databases and **build** their core components.

Questions?

See you in the lab.