Databases

Cloud Computing

Motivation

- Many applications need data to survive after the program exits.
- Keeping data only in memory is not enough; we need to persist it on disk.
- Once data is stored, we need to retrieve and update it efficiently.
- **Database systems** exist to manage stored data reliably and at scale.

CRUD as a Lens

- A simple way to think about database operations is **CRUD**: Create, Read, Update, Delete.
- **`Create`** adds new data, **`Read`** retrieves existing data, **`Update`** changes stored data, and **`Delete`** removes data.
- Different database systems support these operations in different ways and with different performance tradeoffs.
- CRUD is a useful starting point, even though real systems also need richer queries, transactions, and concurrency control.

Databases vs. File Systems

- File systems store bytes in files, but they do not directly support complex queries.
- Applications using plain files often have to implement their own search, update, and indexing logic.
- Database systems are designed around common access patterns such as lookup, filtering, joining, and aggregation.
- This usually makes data access more efficient, especially as data size and query complexity grow.

Why Structure Matters for Efficiency

- If records are stored as an unstructured file, finding one item may require an `O(n)` scan through the data.
- On disk, that can be expensive because reading many blocks is much slower than accessing data already in memory.
- In memory, we already use data structures to organize data so common operations become easier or faster.
- The same basic idea applies on disk: if we choose a good structure for stored data, access can be much more efficient than scanning raw files.
- This is one reason database systems provide more than file storage: they impose useful structure on data to support important access patterns.
- Main question: what's the best structure?

Concurrency as a Core Problem

- Databases are often used by many users and applications at the same time.
- Multiple clients may try to read or update the same data **concurrently**.
- Without coordination, concurrent access can lead to lost updates, inconsistent results, and corrupted state.
- A database system must manage concurrency so shared data remains correct and usable.

Databases as Client-Server Systems

- In many deployments, the database runs as a separate server process.
- Applications act as clients that send queries and updates to the database server.
- The **database engine** is the part of the server responsible for storage, query execution, concurrency control, and recovery.
- This separation lets multiple clients share the same data safely and efficiently.

Single-Machine vs. Distributed Deployment

- The client and database do not always need to run on different machines.
- Embedded systems such as **SQLite** run in the same process or on the same machine as the application.
- In cloud systems, it is more common for clients and database servers to run on different machines.
- Networked deployment makes shared access easier, but it also introduces communication latency and operational complexity.

A “Server” May Be Many Machines

- In cloud deployments, a single logical database server may actually be a coordinated pool of machines.
- Different machines may handle query routing, storage, replication, caching, or failover.
- The system is presented to clients as one service, even though it is implemented by many components.
- This design improves scalability and availability, but it makes coordination more complex.

DATA MODELS

Why a Data Model?

- To store data efficiently, we need to understand how the data will be accessed.
- Different applications need different operations: lookups, scans, updates, joins, or graph traversals.
- The way data is organized should support the operations that matter most.
- A **data model** gives structure to the data and guides how the system stores and accesses it.

Common Types of Data Models

- **Key-value stores** organize data as mappings from keys to values and are optimized for direct lookup by key.
- **Document stores** represent records as flexible structured documents, often using JSON-like formats.
- **Graph databases** represent entities and relationships explicitly, making connected data easier to query.
- The **relational model** organizes data into tables with rows and columns and supports powerful declarative queries.

Key-Value Stores

- A **key-value store** maps each unique key to an associated value.
- This model is simple and fast when the main operation is lookup by key.
- **Redis** is a widely used key-value system, often used for caching, session storage, counters, and queues.
- Example: `user:1234 -> profile data`, `cart:5678 -> shopping cart contents`, `page_views -> counter`.

Strengths and Weaknesses of Key-Value Stores

- Strengths: simple design, fast reads and writes by key, and easy horizontal scaling in many cases.
- They work especially well when access patterns are known in advance and centered on direct key lookup.
- Weaknesses: limited support for complex queries, filtering by attributes, and relationships across records.
- If the application needs rich querying, the simplicity of the model can become a constraint.

Document Databases

- A **document database** stores data as structured documents rather than as rows in a fixed table.
- **Documents** often use JSON-like representations with nested fields and varying structure across records.
- **MongoDB** is a common example of a document database.
- Example: a `users` collection might store one document per user, with fields for profile data, preferences, and addresses.

```
{
  "_id": 1234,
  "name": "Alice",
  "email": "alice@example.com",
  "addresses": [
    { "type": "home", "city":
"Boston" },
    { "type": "work", "city":
"Cambridge" }
  ],
  "preferences": {
    "theme": "dark",
    "notifications": true
  }
}
```

Strengths and Weaknesses of Document Databases

- Strengths: flexible schema, natural fit for hierarchical data, and convenient storage of self-contained application objects.
- They work well when records vary in structure or when related data is usually accessed together in one document.
- Weaknesses: duplicated data is common, and relationships across documents can be harder to manage cleanly.
- For highly connected data or workloads with many joins, the model can become awkward or inefficient.

Graph Databases

- A **graph database** stores data as nodes and edges, with edges representing relationships between entities.
- This model is designed for data where connections are as important as the entities themselves.
- **Neo4j** is a common example of a graph database.
- Example: `Alice -[:FRIEND_OF]-> Bob`, `Bob -[:WORKS_AT]-> Acme Corp`, `Alice -[:LIVES_IN]-> Boston`.
  - Here, `Alice`, `Bob`, `Acme Corp`, and `Boston` are nodes, and labels such as `FRIEND_OF` and `WORKS_AT` name the edges between them.

Strengths and Weaknesses of Graph Databases

- Strengths: natural representation of connected data and efficient traversal of relationships across many hops.
- They work well for social networks, recommendation systems, fraud detection, and knowledge graphs.
- Weaknesses: they are often less natural for tabular reporting and may be unnecessary for simple record-oriented workloads.
- If the application mostly performs simple lookups or aggregations over structured rows, a graph model may add complexity without much benefit.

RELATIONAL DATABASES

What Is a Relational Database?

- A database based on the relational model is called a **relational database**.
- In a relational database, data is organized into **relations**, usually presented as **tables**.
- Relational databases are the systems for which **SQL** became the dominant practical language.

History of the Relational Model

- The relational model was introduced by **Edgar F. Codd** in 1970.
- It was a major shift away from earlier database systems that required programmers to navigate complex record structures manually.
- Codd's key idea was to represent data using relations, which we usually think of as tables.
- This made database systems more **declarative**: users could describe what data they wanted without specifying low-level access paths.
- In the following years, SQL emerged as the dominant practical language for relational databases.

Why the Relational Model Came to Dominate

- It gave a simple and general way to represent many kinds of business and administrative data.
- Its declarative query style made application development easier than low-level record navigation.
- The mathematical foundation made it possible to reason about correctness and optimize queries systematically.
- Over time, strong commercial systems and the rise of SQL made the model the standard choice for many applications.

Core Parts of the Relational Model

- Data is organized into **relations**, which are usually presented as **tables**.
- Each relation has **attributes**, which are the **columns** of the table.
- Each row is a **tuple**, representing one **record** in the relation.
- **Keys** identify tuples and connect relations to one another.

What Is a Relation?

- Formally, a relation comes from set theory.
- If we have sets such as `Students` and `Courses`, their Cartesian product contains all possible pairs `(student, course)`.
- A relation is any subset of that **Cartesian product**: only the pairs that are actually true or relevant.
- Example: if `Enrolled(student, course)` is a relation, then `(Alice, DA510)` is in the relation if Alice is enrolled in DA510.

A Formal Language for Relations

- The relational model is not just a way to store data; it also includes formal ways to manipulate relations.
- These languages describe operations such as **selecting** rows, **projecting** columns, and **combining** relations.
- This formal foundation makes it possible to reason precisely about query meaning and correctness.
- It also provides the basis for database query languages and optimization techniques.

Declarative vs. Imperative Languages

- In an **imperative language**, the programmer specifies step by step how to perform a task.
- In a **declarative language**, the programmer specifies what result is desired.
- Database query languages are valuable because they let users express queries declaratively.
- The database system can then choose an efficient **execution plan** for that query.

Introducing SQL

- **SQL** stands for **Structured Query Language**.
- It is the standard language used to define, query, and update relational databases.
- SQL is largely declarative: users specify the data they want, not the low-level steps to retrieve it.
- In practice, SQL became the main interface through which the relational model was adopted and used.

A Running SQLite Example

- Suppose we have a table called `students`.
- It has columns `id`, `name`, and `major`.
- We can use this one table to illustrate the basic CRUD operations in SQLite.

```
CREATE TABLE students (
  id INTEGER PRIMARY KEY,
  name TEXT,
  major TEXT
);
```

Create in SQLite

- `Create` means inserting new rows into a table.
- In SQLite, this is typically done with `INSERT`.

```
INSERT INTO students (id, name, major)
VALUES (1, 'Alice', 'Data Science');
```

Read in SQLite

- `Read` means retrieving data from the database.
- In SQLite, this is done with `SELECT`.

```
SELECT name, major
FROM students
WHERE id = 1;
```

Update in SQLite

- `Update` means modifying existing rows.
- In SQLite, this is done with `UPDATE`.

```
UPDATE students
SET major = 'Statistics'
WHERE id = 1;
```

Delete in SQLite

- `Delete` means removing rows from a table.
- In SQLite, this is done with `DELETE`.

```
DELETE FROM students
WHERE id = 1;
```

`SELECT` Beyond CRUD

- In practice, `SELECT` is much richer than just “read one row.”
- We can choose specific columns, filter rows, sort results, and limit how much data is returned.
- This is where SQL starts to show the power of declarative querying.

```
SELECT name, major
FROM students
WHERE major = 'Data Science'
ORDER BY name
LIMIT 5;
```

Example: Two Related Tables

- Relational databases are especially useful when data is split across multiple related tables.
- Suppose we also have an `enrollments` table connecting students to courses.

```
CREATE TABLE enrollments (
  student_id INTEGER,
  course TEXT,
  PRIMARY KEY (student_id, course)
);
```

- Now one table stores student information, and another stores which courses each student is taking.

Keys and Constraints

- A **primary key** uniquely identifies each row in a table.
- In the `students` table, `id` is the primary key.
- A **foreign key** is a column in one table that refers to a row in another table.
- In the `enrollments` table, `student_id` is intended to refer to `students.id`.
- **Constraints** are rules the database enforces to keep the data valid and consistent.

Engine-Level Enforcement of Constraints

- In a relational database, constraints are enforced by the database engine itself, not just by application code.
- For example, the engine can reject an insert that duplicates a primary key or a foreign key value that does not match any referenced row.
- This matters because many different applications or users may access the same database.
- Engine-level enforcement keeps the rules centralized, so data integrity does not depend on every client behaving correctly.

Why Normalize?

- A key idea in relational design is **normalization**: separating data so the same fact is not stored repeatedly in many places.
- Repeating data can waste space and, more importantly, create inconsistencies when one copy is updated and another is not.
- By breaking data into related tables, we can store each kind of fact once and connect the tables when needed.
- This improves data integrity, even though it means queries often need ***joins***.

Joins

- A **join** combines rows from multiple tables based on a relationship between them.
- This lets us keep data in separate tables while still querying across them.
- Joins are one of the main reasons the relational model is so useful.

```
SELECT students.name, enrollments.course
FROM students
JOIN enrollments
  ON students.id = enrollments.student_id;
```

Inner Join on a Common Attribute

- An **inner join** matches rows from two tables when a common attribute has the same value in both tables.
- Here, `students.id` and `enrollments.student_id` refer to the same student.
- The database compares those columns and keeps only the pairs of rows where the values match.
- The result combines information from both tables into a single output row.

```
SELECT students.id, students.name,
enrollments.course
FROM students
INNER JOIN enrollments
  ON students.id = enrollments.student_id;
```

DATA MODELING

What Is Data Modeling?

- **Data modeling** is the process of deciding how information in some real-world domain should be represented in a database.
- We want a representation that matches the important facts, supports the queries we care about, and avoids unnecessary duplication.
- In a relational database, this usually means deciding what tables to create, what columns they should have, and how the tables relate.

Entities and Attributes

- A common starting point is to identify the main **entities** in the domain.
- An **entity** is a kind of thing we want to store information about, such as a student, course, instructor, or department.
- **Attributes** are the properties we want to record for each entity, such as a student’s name or a course’s title.
- In a relational design, entities often become tables and attributes often become columns.

Relationships and Cardinality

- We also need to model how entities are related to one another.
- Some relationships are **one-to-one**, some are **one-to-many**, and some are **many-to-many**.
- For example, one department may offer many courses, while many students may enroll in many courses.
- Understanding these patterns helps determine whether we use a foreign key in one table or create a separate relation such as `enrollments`.

From Domain to Schema

- Data modeling is the step where we move from an informal description of a domain to a concrete **database schema**.
- We choose tables, keys, and relationships based on the structure of the domain and the operations we expect to perform.
- A good schema makes the data easier to understand, query, and maintain over time.
- A poor schema makes even simple queries awkward and increases the risk of inconsistency.

Worked Example: A Reddit Clone

- Suppose we want to design a database for a Reddit-like platform.
- We need to store users, communities, posts, and comments.
- This is a good example because it has clear entities, relationships, and common query patterns.
- It also shows why normalized relational designs are useful.

Entities in a Reddit Clone

- `users` stores information about the people using the platform.
- `communities` stores the topic-based forums where posts appear.
- `posts` stores submitted content.
- `comments` stores replies attached to posts, and sometimes replies to other comments.

Relationships in a Reddit Clone

- One user can author many posts and many comments.
- One community can contain many posts.
- One post can have many comments.
- A comment may also refer to another comment, creating a tree of replies.

A Relational Schema for a Reddit Clone

- We might use tables such as `users`, `communities`, `posts`, and `comments`.
- `posts.author_id` refers to `users.id`, and `posts.community_id` refers to `communities.id`.
- `comments.post_id` refers to the post being discussed, and `comments.author_id` refers to the user who wrote it.
- `comments.parent_comment_id` can optionally refer to another comment to represent nested replies.

Why This Design Is Normalized

- User information is stored once in `users`, not repeated on every post and comment.
- Community information is stored once in `communities`, not copied into every post.
- Posts and comments store references to related rows instead of duplicating all of their data.
- This reduces inconsistency and makes updates easier, even though retrieving combined views requires joins.

DATABASE SYSTEM INTERNALS

Transactions

- A **transaction** is a sequence of database operations that should be treated as one logical unit of work.
- For example, transferring money between two accounts should not leave the database in a state where money was removed from one account but not added to the other.
- Transactions help applications group related reads and writes into meaningful actions.
- They are one of the core features that distinguish database systems from simple file storage.

ACID

- Database transactions are often summarized by the **ACID** properties: Atomicity, Consistency, Isolation, Durability.
- **Atomicity** means the whole transaction happens or none of it does.
- **Isolation** means concurrent transactions do not interfere in a way that breaks correctness.
- **Durability** means committed changes survive crashes, and **consistency** captures the idea that transactions preserve the database's rules and invariants.

Concurrency Control

- Because many users and applications may access the database at the same time, transactions can overlap.
- **Concurrency control** is the part of the database system that coordinates those overlapping transactions.
- The goal is to allow useful parallelism while preventing problems such as lost updates, dirty reads, and inconsistent results.
- At a high level, the system tries to make concurrent execution behave as if transactions ran in some safe order.

Recovery and Durability

- Databases must also handle failures such as crashes, power loss, or machine restarts.
- **Recovery mechanisms** ensure that committed transactions are preserved and incomplete transactions are cleaned up correctly.
- This is why databases keep extra information, such as logs, rather than only storing the latest version of the data.
- Durability is not automatic: it is something the database engine works to provide.

Indexes and Physical Structure

- So far we have mostly discussed the logical view of data: tables, rows, joins, and queries.
- A database system also needs physical structures that make those logical operations efficient.
- **Indexes** are one important example: they organize data so the engine can often avoid scanning an entire table.
- This is where implementation details connect back to our earlier question about how structure supports efficient access.

Query Processing and Optimization

- SQL is declarative, so the user states what result is wanted rather than how to compute it.
- The database engine must translate a query into an **execution plan**.
- There may be many possible ways to execute the same query, and some are much faster than others.
- Query optimization is the process of choosing an efficient plan based on the query, the schema, and available indexes.

ORMS

What Is an ORM?

- ORM stands for **Object-Relational Mapping**.
- An ORM is a library or framework that helps application code interact with a relational database using objects, classes, or language-native data structures.
- Instead of writing raw SQL for every operation, developers can often work with higher-level abstractions.
- The ORM translates those operations into SQL queries sent to the database.

Why Use an ORM?

- Many applications are written in object-oriented or object-adjacent languages, so developers naturally think in terms of objects and methods.
- An ORM can reduce repetitive SQL boilerplate for common operations such as loading, creating, updating, and deleting records.
- ORMs can also help centralize schema definitions, relationships, and validation rules in application code.
- The basic goal is convenience: make it easier for application code and relational data to work together.

From ORM to REST API

- Many applications do not expose the database directly to users.
- Instead, the database sits behind an application service.
- The service exposes HTTP endpoints, receives requests, runs database operations, and returns responses.
- This is a common pattern for building database-backed web applications.

FastAPI as the Service Layer

- FastAPI is a Python framework for building HTTP APIs.
- In a database-backed service, FastAPI handles routing, request parsing, response formatting, and API documentation.
- The database layer handles persistence, queries, constraints, and transactions.
- Keeping these responsibilities separate makes the system easier to reason about.

REST Endpoints and Database Operations

- REST APIs often map HTTP methods to application-level CRUD operations.
- `POST /posts` can create a new post.
- `GET /posts/{post_id}` can read one post.
- `PATCH /posts/{post_id}` can update a post.
- `DELETE /posts/{post_id}` can delete a post.

Request Flow in a FastAPI App

- A client sends an HTTP request to a FastAPI endpoint.
- FastAPI validates the request data and calls a Python function for that route.
- That function uses an ORM or database access layer to query or modify the relational database.
- The result is converted into an HTTP response and returned to the client.

Example: Creating a Post

```
@app.post("/posts")
def create_post(post: PostCreate, db: Session =
Depends(get_db)):
    db_post = Post(title=post.title, body=post.body,
author_id=post.author_id)
    db.add(db_post)
    db.commit()
    db.refresh(db_post)
    return db_post
```

- The endpoint receives structured request data.
- The ORM object represents a row that will be inserted into the `posts` table.
- `commit()` makes the change durable in the database.

What the REST Service Adds

- It hides direct database access from clients.
- It centralizes validation, authentication, authorization, and application rules.
- It can expose a stable API even if the internal schema changes.
- It also becomes the place where database errors must be translated into useful HTTP responses.

AWS DEPLOYMENTS

Managed vs. Self-Managed Databases on AWS

- On AWS, the default choice for many teams is a **managed database service** such as Amazon **RDS** or Amazon **Aurora**.
- With a managed service, AWS handles much of the operational work: provisioning, patching, backups, monitoring integrations, and failover features.
- The alternative is self-managing a database on EC2, which gives more control but also more operational responsibility.
- The core tradeoff is control versus operational simplicity.

RDS and Aurora

- Amazon **RDS** is AWS's managed relational database service for engines such as PostgreSQL, MySQL, MariaDB, Oracle, SQL Server, and Db2.
- Amazon **Aurora** is an AWS-designed relational database engine compatible with PostgreSQL and MySQL.
- Both services let teams run relational databases without managing the underlying database servers directly.
- Aurora adds a cloud-native storage and replication architecture designed for higher scalability and availability.

Network Placement

- A cloud database should usually not be exposed directly to the public internet.
- In AWS, databases are placed inside a VPC and access is controlled with subnet placement and security groups.
- A common pattern is to let application servers connect to the database while blocking direct access from the outside world.
- This makes network design part of database security, not just infrastructure plumbing.

High Availability

- Production databases need a plan for machine or Availability Zone failure.
- Amazon RDS and Aurora support high-availability deployment patterns such as Multi-AZ configurations.
- The goal is to keep the database available if part of the infrastructure fails.
- High availability usually costs more, but it reduces the risk that one machine failure takes down the application.

Backups and Recovery

- Backups are a core part of database operations, not an optional add-on.
- Managed AWS database services support features such as automated backups, snapshots, and point-in-time recovery.
- Recovery planning matters because the important question is not just “do we have backups?” but “can we restore to the state we need?”
- Backup retention and recovery objectives should be chosen based on how much data loss and downtime the application can tolerate.