What you will learn

In this lecture you will learn:

- The **limitations** of the **relational data model**.

- What a **distributed database** is.

- How data is **distributed** across different machines.

- The **availability-consistency** trade-off (CAP theorem).

- The main characteristics of **NoSQL databases**.

- The families of NoSQL databases.
Relational data model limitations: impedance mismatch

**Definition (Impedance mismatch)**

**Impedance mismatch** refers to the challenges encountered when one needs to map objects used in an application to tables stored in a relational database.

---

Diagram:

- **Application**
  - objects
  - Mapping
  - **Relational database**
    - tables
    - **Book**
      - isbn
      - title
    - **Publisher**
      - publisher_id
      - name
      - country
    - **Author**
      - author_id
      - first_name
      - last_name
      - country
    - **Book_author**
      - author_id
      - isbn
    - **Book**
      - isbn
      - title
      - publisher_id
Impedance mismatch: solutions

Object-oriented databases

- Data is stored as **objects**.
- Object-oriented applications save their objects as they are.
- **Examples.** ConceptBase, Db4o, Objectivity/DB.

Disadvantage

- Not as popular as relational database systems.
- Requires familiarity with object-oriented concepts.
- No standard query language.
Impedance mismatch: solutions

Object relational mappers (ORM)
- Use of libraries that map objects to relational tables.
- The application manipulates objects.
- The ORM library translates object operations into SQL queries.
- **Examples.** SQLAlchemy, Hibernate, Sequelize.

Disadvantage
- **Abstraction.** Weak control on how queries are translated.
- **Portability.** Each ORM has a different set of APIs.
Limitations of the relational model: graph data

Normalization

- In a relational databases, tables are **normalized**.
- Data on **different entities** are kept in **different tables**.
- This reduces **redundancy** and guarantees **integrity**.

- In a **normalized** relational database, links between entities are expressed with **foreign key constraints**.
- Need to join different tables (**expensive** operation).

<table>
<thead>
<tr>
<th>Book</th>
<th>join</th>
<th>Book_author</th>
<th>join</th>
<th>Author</th>
</tr>
</thead>
<tbody>
<tr>
<td>isbn</td>
<td></td>
<td>author_id</td>
<td></td>
<td>author_id</td>
</tr>
<tr>
<td>title</td>
<td></td>
<td>isbn</td>
<td></td>
<td>first_name</td>
</tr>
<tr>
<td>publisher_id</td>
<td></td>
<td></td>
<td></td>
<td>last_name</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>country</td>
</tr>
</tbody>
</table>
Limitations of the relational model: data distribution

Objective of a relational database system

- Privilege data **integrity** and **consistency**.
- Different mechanisms to ensure integrity and consistency.
  - Primary and foreign key constraints.
  - Transactions.
- Mechanisms to enforce data integrity and consistency have a **cost**.
  - Manage transactions.
  - Check that new data complies with the given integrity constraints.
- Things get worse in **distributed databases**.
  - Data is distributed across several machines.
  - Join operations become very expensive.
  - Integrity mechanisms become very expensive.
Distributed database

Definition (Distributed database)

A **distributed database** is one where data is stored across several **machines**, a.k.a, **nodes**.

Shared-nothing architecture

- Each node has its own CPU, memory and storage.
- Nodes only share the network connection.

Pros/cons of a distributed database

- Allows storage and management of large volumes of data. 😊
- Far more complex than a single-server database. 😞
Distributed database

All users access the database through one global schema or database. The global schema is simply the union of all the local database schemas. It is difficult in most organizations to force a homogeneous environment, yet heterogeneous environments are much more difficult to manage. As listed previously, there are many variations of heterogeneous distributed database environments. In the remainder of the chapter, however, a heterogeneous environment will be defined by the following characteristics (as depicted in Figure 13-3):

- Data are distributed across all the nodes.
- Different DBMSs may be used at each node.
- Some users require only local access to databases, which can be accomplished by using only the local DBMS and schema.
- A global schema exists, which allows local users to access remote data.

It is important to note that different variants of cloud-based databases (discussed in Chapter 8) are often implemented in a distributed way. The cloud service providers use a variety of approaches to achieve the benefits of a distributed database architecture. These features are, however, typically hidden from most users and developers. Commitments regarding the geographic distribution of data might become part of a service level agreement between the customer and a service provider. These details are, however, beyond the scope of this book.

Objectives and Trade-Offs

A major objective of distributed databases is to provide ease of access to data for users at many different locations. To meet this objective, the distributed database system must provide **location transparency**, which means that a user (or user program) using data for querying or updating need not know the location of the data. Any request to retrieve or update data from any site is automatically forwarded by the system to the site or sites related to the processing request. Ideally, the user is unaware of the distribution of data, and all data in the network appear as a single logical database stored at one site. In this ideal case, a single query can join data from tables in multiple sites as if the data were all in one site.
Distributing data: when?

Small-scale data
- Data distribution is not a good option when the **data scale is small**.
- With **small-scale data**, the performances of a distributed database are **worse** than a single-server database.
  - **Overhead**. We lose more time distributing and managing data than retrieving it.

Large-scale data
- If the data does not fit in a single machine, data distribution is the only option left.
- Distributed databases allow **more concurrent database requests** than single-server databases.
Distributing data: how?

Data distribution options

- **Replication.** Multiple copies of the same data stored on different nodes.
- **Sharding.** Data partitions stored on different nodes.
- **Hybrid.** Replication + Sharding.

Properties

- **Location transparency:** applications do not have to be aware of the location of the data.
- **Replication transparency:** applications do not need to be aware that the data is replicated.
Replication

- The same piece of data is replicated across different nodes.
- Each copy is called a **replica**.
- **Replication factor.** The number of nodes on which the data is replicated.
Replication

Advantages

- **Scalability.** Multiple nodes can serve queries on the same data.
- **Latency.** Queries can be served by geographically proximate nodes.
- **Fault tolerance.** The database keeps serving queries even if some nodes fail.

Disadvantages

- **Storage cost.** Storage is used to keep multiple copies of the same data.
- **Consistency.** All replicas must be kept in sync.
Replication

Replica consistency

When a replica is updated, the other replicas must be updated as well.
Replication

Synchronous updates

- Updates are propagated immediately to the other replicas.
- **Small inconsistency window.** The replicas will be inconsistent for a short interval of time. 😊
- If updates are frequent, the database might be too busy propagating updates than serving queries. 😞

Asynchronous updates

- Updates are propagated at regular intervals.
- More efficient when updates are frequent. 😊
- Long inconsistency window. 😞
**Replication**

**Master-slave replication**

- **Write** operations are only possible on the **master node**.
- The **master node** propagates the updates to the **slave nodes**.
- **Read** operations are served by both the master and the slave nodes.
Replication

**Master-slave replication**

- Prevents **write conflicts**. 😊
  - Only one replica is written at any given time.
- **Single point of failure**. 😞
  - If the master fails, write operations are unavailable.
  - Algorithms exist to **elect** a new master.
- **Read conflicts** are possible. 😞
Master-slave replication read conflict

Two **read** operations on the **same data** might return **different values**.

**Write**: update (Department, budget=500,000)

**Read**: select (Department, budget)

<table>
<thead>
<tr>
<th>Department</th>
<th>code</th>
<th>nameD</th>
<th>budget</th>
</tr>
</thead>
<tbody>
<tr>
<td>Administration</td>
<td>14</td>
<td>Administration</td>
<td>300,000</td>
</tr>
<tr>
<td>Education</td>
<td>25</td>
<td>Education</td>
<td>150,000</td>
</tr>
<tr>
<td>Finance</td>
<td>62</td>
<td>Finance</td>
<td>600,000</td>
</tr>
<tr>
<td>Human Resources</td>
<td>45</td>
<td>Human Resources</td>
<td>150,000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Department</th>
<th>code</th>
<th>nameD</th>
<th>budget</th>
</tr>
</thead>
<tbody>
<tr>
<td>Administration</td>
<td>14</td>
<td>Administration</td>
<td>500,000</td>
</tr>
<tr>
<td>Education</td>
<td>25</td>
<td>Education</td>
<td>150,000</td>
</tr>
<tr>
<td>Finance</td>
<td>62</td>
<td>Finance</td>
<td>600,000</td>
</tr>
<tr>
<td>Human Resources</td>
<td>45</td>
<td>Human Resources</td>
<td>150,000</td>
</tr>
</tbody>
</table>
Replication

Peer-to-peer replication

- **Read** and **write** operations are possible on **any node**.

![Diagram showing peer-to-peer replication with nodes A, B, and C. Each node has read and write operations indicated.]
Replication

Peer-to-peer replication

- No single point of failure. 😊
- Write and read conflicts are possible. 😞
Sharding

- Data is partitioned into balanced, non-overlapping **shards**.
- Shards are distributed across the nodes.
Sharding

Advantages

- **Load balance.** Data can be uniformly distributed across nodes.
- **Inconsistencies** cannot arise (non-overlapping shards).

Disadvantages

- When a node fails, all its partitions are lost.
- Join operations might need to be performed across nodes.
- When data is added, shards might need to be rebalanced.
Combining replication and sharding
Consistency in distributed databases

Replication consistency
Keeping in sync all replicas of the same data.

Cross-record consistency
Ensuring the coherence of data in related records. Related records might be on different nodes.

<table>
<thead>
<tr>
<th>codeD</th>
<th>nameD</th>
<th>budget</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>Administration</td>
<td>300,000</td>
</tr>
<tr>
<td>25</td>
<td>Education</td>
<td>150,000</td>
</tr>
</tbody>
</table>

UPDATE Department
SET codeD=15
WHERE codeD=14
Definition (Distributed transactions)

A distributed transaction is a sequence of read/write operations that are applied on data that reside on multiple nodes and are executed as an atomic operation.

We need to update the codeD of each employee in department 14.
Consistency in distributed databases

Distributed transaction

- The nodes need to **coordinate** before committing the transaction operations on their data.
- The coordination requires an exchange of messages between the **transaction managers** on different nodes.
Distributed databases and NoSQL

Data distribution

Consistency in distributed databases

Distributed transaction

- Data being manipulated by a transaction is locked.
- Locked data is unavailable for both read and write operations.
- Locking guarantees the consistency of the database.
- Locking reduces the availability of the database.

<table>
<thead>
<tr>
<th>code</th>
<th>name</th>
<th>budget</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>Administration</td>
<td>300,000</td>
</tr>
<tr>
<td>25</td>
<td>Education</td>
<td>150,000</td>
</tr>
</tbody>
</table>

UPDATE Department
SET codeD=15
WHERE codeD=14

UPDATE Employee
SET codeD=15
WHERE codeD=14

ready to commit?
The CAP theorem

Consistency (C), Availability (A), Partition tolerance (P)

- **Consistency.** Replicas are in sync and related records are coherent across all nodes.
- **Availability.** A database can still execute read/write operations when some nodes fail.
- **Partition tolerance.** The database can still operate when a network partition occurs.
The CAP theorem

Theorem (CAP, Brewer 1999)

Given the three properties of consistency, availability and partition tolerance, a networked shared-data system can have at most two of these properties.

Proof

Suppose that the system is partition tolerant (P). When a network partition occurs, we have two options.

1. **Allow write operations.** This makes the database available (A), but not consistent (C).
   - Some of the replicas might not be synced due to the network partition.

2. **Disable write operations.** This makes the database consistent (C) but not available (A).
The CAP theorem

Theorem (CAP, Brewer 1999)

*Given the three properties of consistency, availability and partition tolerance, a networked shared-data system can have at most two of these properties.*

Proof

- The only way that we can have a consistent (C) and available (A) database is when network partitions do not occur.
- But if we assume that network partitions never occur, the system is not partition tolerant (P).
Consistency vs Availability

- Relational databases favor **consistency** over **availability**.
  - They take a **transactional approach** to data consistency.
- NoSQL databases favor **availability** over **consistency**.
  - In many contexts strong consistency is not necessary.

Alice does not see Bob’s post between $t_1$ and $t_2$. Is it really an issue?
ACID vs BASE

ACID (strong consistency)

- Atomicity (A). “All or nothing”.
- Consistency (C). From a consistent state to a consistent state.
- Isolation (I). Serializability of transactions.
- Durability (D). Upon commit, all the updates are permanent.

BASE (availability)

- Basic Availability (BA). The database appears to work most of the time.
- Soft state (S). Write and read inconsistencies can occur.
- Eventually consistent (E). The database will be consistent at some point.
NoSQL databases

NoSQL: interpretations of the acronym

- *Non SQL*: strong opposition to SQL.
- *Not only SQL*: NoSQL and SQL coexistence.

Goals

- Address the **object-relational impedance mismatch**.
- Provide better scalability for **distributed databases**.
- Provide a better modeling of **semi-structured data**.
NoSQL databases

**Families**

- **Key-value** databases.
- **Document-oriented** databases.
- **Column-oriented** databases.
- **Graph** databases.

The first three families use the notion of **aggregate** to model the data.

- They differ in how the aggregates are organized.

**Graph** databases are somewhat **outliers**.

- They were not conceived for data distribution in mind.
- They were born ACID-compliant.

❗ There is not a single NoSQL database and there is not a “NoSQL” query language.
An **aggregate** is a data structure used to store the data of a specific entity.

- In that, it is similar to a row in a relational table.

We can **nest** an aggregate into another aggregate.

- This is a huge difference from a row in a relational table.

An aggregate is a **unit of data** for **replication** and **sharding**.

- All data in an aggregate will never be split across two shards.
- All data in an aggregate will always be available on one node.
- Unlike a relational database, we can control how data is distributed.
**Aggregate vs relational row**

**Denormalized table**

- In a relational database, the following table would not be in **first normal form**.
- The column *categories* contains a list of values.
  - Searching for all products in category *kitchen* would be hard with SQL.

<table>
<thead>
<tr>
<th>article_id</th>
<th>name</th>
<th>producer</th>
<th>categories</th>
</tr>
</thead>
<tbody>
<tr>
<td>234543</td>
<td>Bamboo utensil spoon</td>
<td>KitchenMaster</td>
<td>home, kitchen, spatulas</td>
</tr>
</tbody>
</table>

⚠️ In a relational database, we can address this problem by **normalizing** the table.
First normal form

- The following table is in **first normal form**.
- But we introduced **redundancy**.
  - What if we update the producer name of the article 234543?
  - In a distributed database, the rows corresponding to this article might be on different nodes.

<table>
<thead>
<tr>
<th>article_id</th>
<th>name</th>
<th>producer</th>
<th>categories</th>
</tr>
</thead>
<tbody>
<tr>
<td>234543</td>
<td>Bamboo utensil spoon</td>
<td>KitchenMaster</td>
<td>home</td>
</tr>
<tr>
<td>234543</td>
<td>Bamboo utensil spoon</td>
<td>KitchenMaster</td>
<td>kitchen</td>
</tr>
<tr>
<td>234543</td>
<td>Bamboo utensil spoon</td>
<td>KitchenMaster</td>
<td>spatulas</td>
</tr>
</tbody>
</table>

💡 We can **further normalize** the table to avoid redundancy.
Aggregate vs relational row

Second normal form

- To avoid redundancy, we split the table into three tables in second normal form.
- In a distributed database, the rows in these tables might be on different nodes.
- We might need cross-node join operations, which are very expensive.

<table>
<thead>
<tr>
<th>article</th>
<th>article_category</th>
<th>category</th>
</tr>
</thead>
<tbody>
<tr>
<td>article_id</td>
<td>name</td>
<td>producer</td>
</tr>
<tr>
<td>234543</td>
<td>Bamboo utensil spoon</td>
<td>KitchenMaster</td>
</tr>
<tr>
<td>234543</td>
<td></td>
<td></td>
</tr>
<tr>
<td>234543</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Aggregate vs relational row

**Aggregate**

- In an **aggregate**, list of values are **allowed**.
- Searching for all products in category *kitchen* is supported.

```json
{
    "article_id": 234543,
    "name": "Bamboo utensil spoon",
    "producer": "KitchenMaster",
    "categories": ["home", "kitchen", "spatulas"]
}
```

⚠️ All data in an aggregate is never split across different nodes.
Denormalization is allowed in the aggregate.

Data that are queried together are stored in the same node.

```json
{
    "code_employee": 12353,
    "first_name": "John",
    "last_name": "Smith",
    "salary": 50000,
    "position": "Assistant director",
    department: {
        "dept_code": 12,
        "dept_name": "Accounting",
        budget: 120000
    }
}
```
• Aggregates are **schemaless**.
• Aggregates might not have the same attributes.

```json
{
  "code_employee": 12353,
  "first_name": "John",
  "last_name": "Smith",
  "salary": 50000,
  "position": "Assistant director",
  department: {
    "dept_code": 12,
    "dept_name": "Accounting",
    budget: 120000
  }
}

{
  "code": 345321,
  "first_name": "Jennifer",
  "last_name": "Green",
}
```

⚠️ We don’t need to fix a rigid the schema. NULL values are avoided.
{  
  "code_employee": 12353,  
  "first_name": "John",  
  "last_name": "Smith",  
  "salary": 50000,  
  "position": "Assistant director" ,  
  "departments": [    
  {    
    "dept_code": 12,  
    "dept_name": "Accounting",  
    "budget": 120000  
  },    
  {    
    "dept_code": 145,  
    "dept_name": "HR",  
    "budget": 250000  
  }  
  ]  
}
We can update **atomically** the salary of an employee. How would we represent the same in a relational database?
We use a **denormalized table** (same as aggregate).

However, we have no guarantees that the rows relative to the employee John Smith will be stored in the same node.

<table>
<thead>
<tr>
<th>code_emp</th>
<th>first_name</th>
<th>last_name</th>
<th>salary</th>
<th>position</th>
<th>dept_code</th>
<th>dept_name</th>
<th>budget</th>
</tr>
</thead>
<tbody>
<tr>
<td>234543</td>
<td>John</td>
<td>Smith</td>
<td>50000</td>
<td>Assistant director</td>
<td>12</td>
<td>Accounting</td>
<td>120000</td>
</tr>
<tr>
<td>234543</td>
<td>John</td>
<td>Smith</td>
<td>50000</td>
<td>Assistant director</td>
<td>145</td>
<td>HR</td>
<td>250000</td>
</tr>
</tbody>
</table>

The update of the salary of a single employee might be a **cross-node operation**.
Updating the information on a department is a non-atomic operation
We’ll see how to alleviate this problem when we introduce MongoDB.
Aggregator-based NoSQL databases

- Aggregates are **schemaless**.
  - No need to adhere to a rigid schema.
  - Flexible evolution of the database.

- Normalization is not required.
  - We accept some **redundancies** in exchange of faster queries.
  - Remember: storage hardware is **cheap** today.

- All data in an aggregate is stored in a **single node**.
  - With aggregates, we are in control of how the data is distributed.

- In general, updates on an aggregate are **atomic operations**.
  - If an update entails many write operations, either all are executed or none.

- Cross-aggregate updates are **not guaranteed** to be atomic.
  - Multi-aggregate transactions might be supported and used if necessary.
Key-value databases

Idea

Data are modeled as **key-value pairs**.

- **Key**: alphanumeric string, usually auto-generated by the database.
- **Value**: an aggregate.
- **Query**: get an aggregate given its key.

![Diagram of key-value pairs]
Key-value databases

**Idea**
- Data is partitioned based on the key.
- Partitions are distributed across different nodes.
- Little to no checks on integrity constraints.
- **Goal.** High scalability and fast read/write queries.

![Diagram showing key-value databases with keys and associated products](image_url)
Key-value databases

Application scenarios

**Scenario 1. Session store.**

- A Web application starts a session when a user logs in.
- The application stores **session data** in the database.
  - User profile information, messages, personalized themes...
- Each session is assigned a **unique identifier** (the key).
- Session data is only queried by the identifier.
Scenario 2. Shopping cart.

- An e-commerce website may receive billions of orders in seconds.
- Each shopping cart has a **unique identifier** (the key).
- Shopping cart data is only queried by the identifier.
- Shopping cart data can be easily replicated to handle node failures.
Key-value databases

Existing key-value databases

- **Amazon DynamoDB.** One of the first NoSQL databases.
- **Riak.**
- **Redis.** Possibility of tuning data persistence.
- **Voldemort.**
Document-oriented databases

**Idea**

- Data is modeled as **key-value pairs**, and searching aggregates based on their **attribute values** is supported.

```
Database
Collection
Document
product_id: 12234345
name: “Bamboo utensil spoon”
categories: [“home”, “kitchen”, “spatulas” ]

Document

product_id: 98761
name: “Mini round cocotte”
categories: [“home”, “kitchen”, “dining” ]
```

It is possible to search for all products in category *kitchen*. 
Document-oriented databases

Existing document-oriented databases

- MongoDB, CouchDB, OrientDB.

Database
Collection
Document
product_id: 12234345
name: “Bamboo utensil spoon”
categories: [“home”, “kitchen”, “spatulas” ]

Document
product_id: 98761
name: “Mini round cocotte”
categories: [“home”, “kitchen”, “dining” ]
Column-oriented databases

Idea

- Similar to document-oriented database but an aggregate can be broken into smaller data units called columns.
**Idea**

- Columns can be organized into **column families**.
- Columns in the same family are stored on the same node.

### Example

**Row Identifier:** 1234

**Column Family:**
- **Profile**
  - `codeE`: 1
  - `first`: Joseph
  - `last`: Bennet
  - `position`: Office assistant

**Column Family:**
- **Department**
  - `codeD`: 14
  - `nameD`: Administration
  - `budget`: 30000
Column-oriented databases

Idea

- The value of a column can be an aggregate (wide column).

Row identifier: 23342

- Column family: Department
  - Column key: code
    - Value: 14
  - Column key: name
    - Value: Administration
  - Column key: budget
    - Value: 3000

- Column family: Employees
  - Column key: code
    - Value: 1234 [Joseph Bennet, Office Assistant, 55000]
    - Value: 2345 [Michael Watson, Team Leader, 80000]
    - Value: 3452 [Jennifer Young, Assistant Director, 120000]
Column-oriented databases

Existing column-oriented databases

- **Cassandra, HBase, BigTable (Google)**.
Graph databases

Idea

- Their data model is optimized for storing and retrieving **graph data**.
- Relationships are **first-class citizens**.
  - In relational databases they are implicit in **foreign key constraints**.
  - In aggregate-based NoSQL stores, they are represented with nested aggregates or references.
- Existing graph databases: Neo4j, InfiniteGraph, AllegroGraph.
NoSQL databases: conclusions

Polyglot persistence

- NoSQL databases are **not** going to replace relational databases.
- Use of different data storage technologies based on the data type.
- This is called **polyglot persistence**.