# Big Data

## Lecture 5 – Distributed and NoSQL databases

**Gianluca Quercini**

gianluca.quercini@centralesupelec.fr

Centrale DigitalLab, 2021

CENTRALE MARSEILLE

# 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.

# Introduction

## What we've seen so far

- Hadoop and Spark as **distributed data processing** frameworks.
- Data from **text files** stored in a **distributed file system** (HDFS).

## What we're going to see

- Data can be stored and managed by **database systems**.
- As opposed to a **file system**, a **database** provides:
  - Data model and query language.
  - Indexing and integrity constraints.
  - Fine-grained security mechanisms.
  - Concurrency control.
  - Backup and recovery.
- The most popular database systems are based on the **relational data model** (Source).

# The relational data model

☞ In the **relational model**, a database is a collection of **tables**, or **relations**.

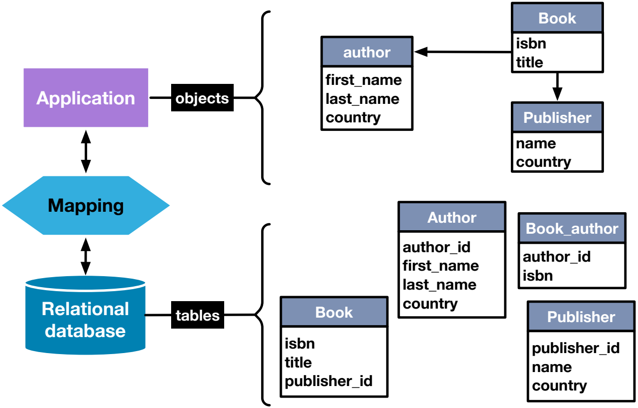
- A **row** in a table (or, a **tuple** in a **relation**) describes an **entity**.
- A **column** in a table (or, an **element** in a **tuple**) represents an **attribute** of an entity.
- A **relationship** between two entities is expressed as common values in one or more columns of their respective tables.
- The relational model provides an *open-ended* collection of **scalar types** (e.g., *boolean*, *integer* . . . ).
  - Open-ended: users are allowed to define custom types.

☞ The values in a given column must have the **same type**.

# 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.



# 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

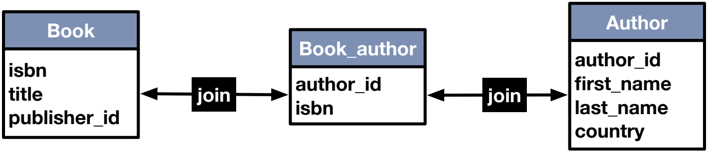
- **Abstraction.** Weak control on how queries are translated.
- **Portability**. Each ORM has a different set of APIs.

# 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).



# 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**.

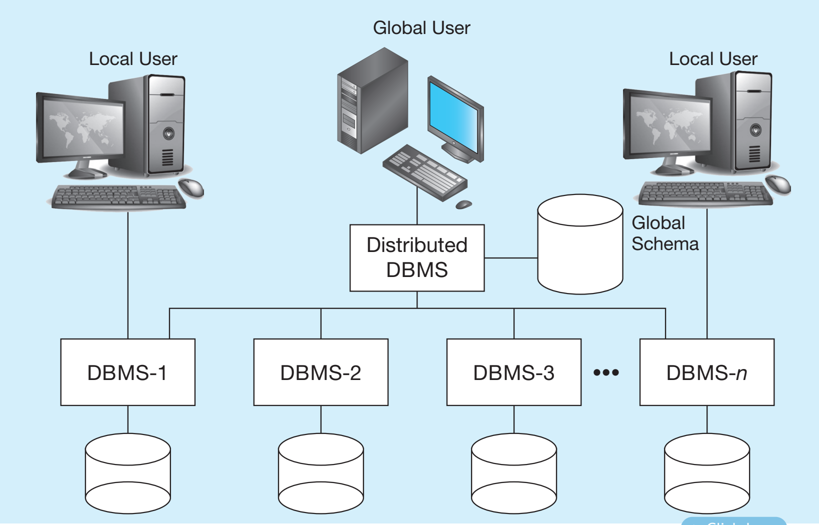
## Shared-nothing architecture

- 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




Click here

# 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.

**A**

| Department | | |
|---|---|---|
| **codeD** | **nameD** | **budget** |
| 14 | Administration | 300,000 |
| 25 | Education | 150,000 |
| 62 | Finance | 600,000 |
| 45 | Human Resources | 150,000 |

**B**

| Department | | |
|---|---|---|
| **codeD** | **nameD** | **budget** |
| 14 | Administration | 300,000 |
| 25 | Education | 150,000 |
| 62 | Finance | 600,000 |
| 45 | Human Resources | 150,000 |

**C**

| Department | | |
|---|---|---|
| **codeD** | **nameD** | **budget** |
| 14 | Administration | 300,000 |
| 25 | Education | 150,000 |
| 62 | Finance | 600,000 |
| 45 | Human Resources | 150,000 |

# 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.

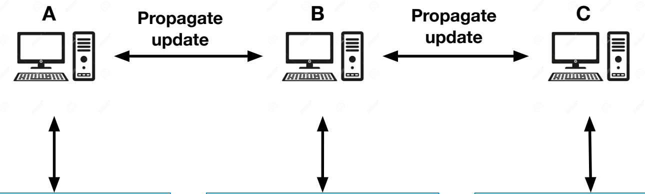
## Disadvantages

- **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.




**UPDATE Department**
**SET budget=500000**
**WHERE codeD=14**

**Department** (A)

| codeD | nameD | budget |
|---|---|---|
| 14 | Administration | **500,000** |
| 25 | Education | 150,000 |
| 62 | Finance | 600,000 |
| 45 | Human Resources | 150,000 |

**Department** (B)

| codeD | nameD | budget |
|---|---|---|
| 14 | Administration | 300,000 |
| 25 | Education | 150,000 |
| 62 | Finance | 600,000 |
| 45 | Human Resources | 150,000 |

**Department** (C)

| codeD | nameD | budget |
|---|---|---|
| 14 | Administration | 300,000 |
| 25 | Education | 150,000 |
| 62 | Finance | 600,000 |
| 45 | Human Resources | 150,000 |

# 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. ☹

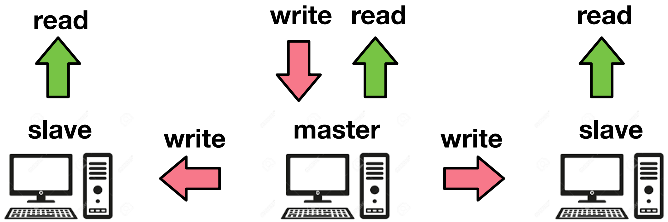
## Asynchronous updates

- Updates are propagated at regular intervals.
- More efficient when updates are frequent. ☺
- Long inconsistency window. ☹

# Replication

## Master-slave replication

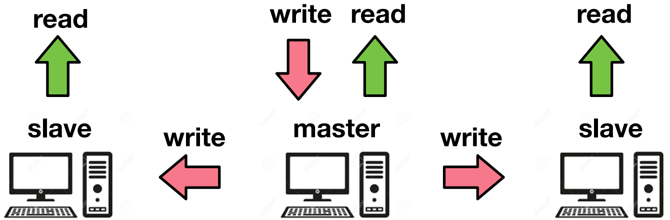
- **Write** operations are only possible on the **master node**.
- The **master node** propagates the updates to the **slave nodes**.
- **Read** operations are served by both the master and the slave nodes.



# 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. ☹

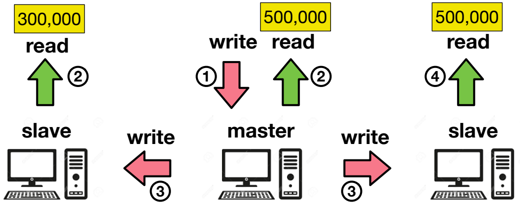


# Replication

## 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)




| Department | | |
|---|---|---|
| **codeD** | **nameD** | **budget** |
| 14 | Administration | **300,000** |
| 25 | Education | 150,000 |
| 62 | Finance | 600,000 |
| 45 | Human Resources | 150,000 |

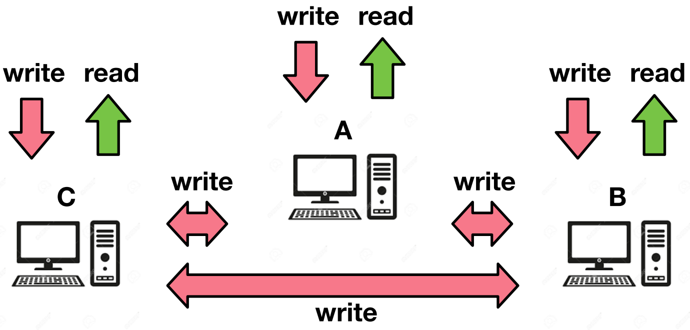
| Department | | |
|---|---|---|
| **codeD** | **nameD** | **budget** |
| 14 | Administration | **500,000** |
| 25 | Education | 150,000 |
| 62 | Finance | 600,000 |
| 45 | Human Resources | 150,000 |

| Department | | |
|---|---|---|
| **codeD** | **nameD** | **budget** |
| 14 | Administration | **500,000** |
| 25 | Education | 150,000 |
| 62 | Finance | 600,000 |
| 45 | Human Resources | 150,000 |

# Replication

## Peer-to-peer replication

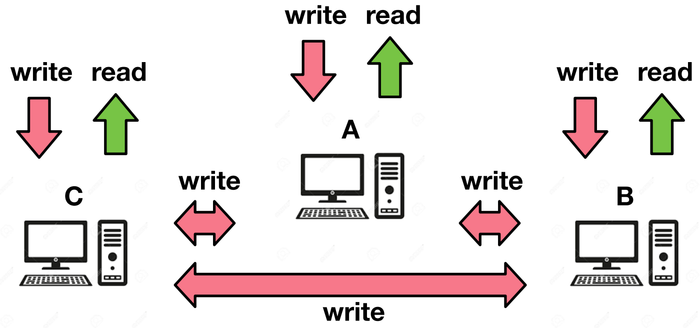
- **Read** and **write** operations are possible on **any node**.



# Replication

## Peer-to-peer replication

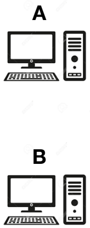
- No single point of failure. ☺
- Write and read conflicts are possible. ☹



# Sharding

## Sharding

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




**Department**

| codeD | nameD | budget |
|---|---|---|
| 14 | Administration | 300,000 |
| 25 | Education | 150,000 |

**Employee**

| codeE | last_name | codeD |
|---|---|---|
| 1 | Bennet | 14 |
| 2 | Doe | 62 |
| 3 | Fisher | 25 |
| 4 | Green | 62 |

**Employee**

| codeE | last_name | codeD |
|---|---|---|
| 5 | Russel | 25 |
| 6 | Smith | 62 |
| 7 | Watson | 14 |
| 8 | Young | 45 |

**Department**

| codeD | nameD | budget |
|---|---|---|
| 62 | Finance | 600,000 |
| 45 | Human Resources | 150,000 |

# Sharding

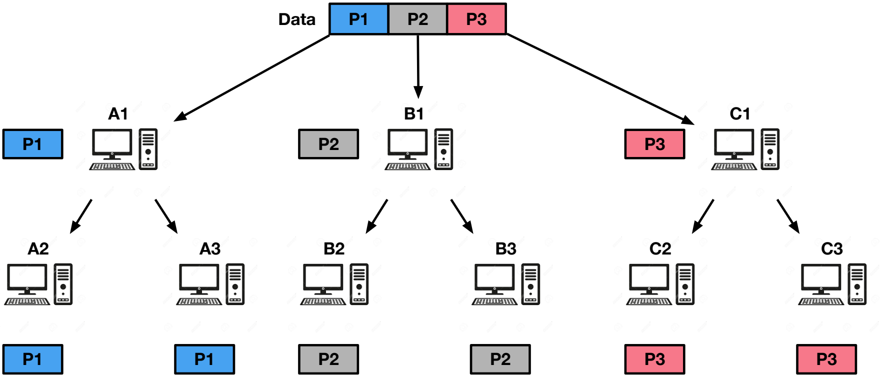
## Advantages

- **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: first definition

## Definition (Consistency)

A database is **consistent** if the data respect all the **integrity constraints** imposed by the database administrator.

| Employee | | |
|---|---|---|
| **codeE** | **last_name** | **codeD** |
| 1 | Bennet | 14 |
| 2 | Doe | 62 |
| 3 | Fisher | 25 |
| 4 | Green | 62 |

```
UPDATE Department
SET codeD=15
WHERE codeD=14
```

| Department | | |
|---|---|---|
| **codeD** | **nameD** | **budget** |
| 14 | Administration | 300,000 |
| 25 | Education | 150,000 |

- **Transactions** are used to keep a database consistent.

## ACID

**A**tomicity, **C**onsistency, **I**solation, **D**urability.

# Consistency in distributed databases

- **Distributed transactions** are used to keep a distributed database consistent.
- **Transaction managers** in all the nodes involved in the transaction need to communicate before committing.
- This communication is expensive.


| codeD | nameD | budget |
|---|---|---|
| 14 | Administration | 300,000 |
| 25 | Education | 150,000 |

**Employee**

| codeE | last_name | codeD |
|---|---|---|
| 1 | Bennet | 14 |
| 2 | Doe | 62 |
| 3 | Fisher | 25 |
| 4 | Green | 62 |

```
UPDATE Department
SET codeD=15
WHERE codeD=14
```

```
UPDATE Employee
SET codeD=15
WHERE codeD=14
```

# Consistency vs Availability

- 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.

### Relational vs NoSQL databases

- Relational databases favor **consistency** over **availability**.
  - **ACID**-compliant databases.
- NoSQL databases favor **availability** over **consistency**.
  - **BASE**: **B**asic **A**vailability, **S**oft state, **E**ventually consistent.

# Consistency: second definition

**Definition (Consistency)**

A (distributed) database is **consistent** if reads and updates behave as if there were a single copy of the data. (▸ Source).

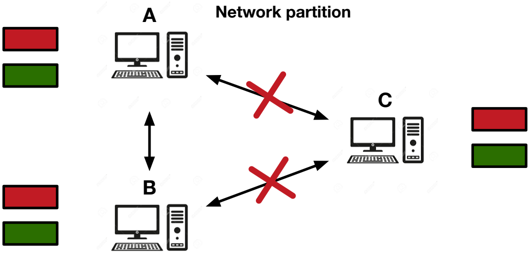
- This second definition of consistency refers to **replication consistency**.
- Enforcing (strong) consistency creates problems with availability.
- What to do when the nodes of a cluster cannot communicate (network issues)?

The **CAP theorem** describes the relation between **consistency**, **availability** and **partition tolerance**.

# The CAP theorem

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

- **Consistency.** As intended by the **second definition**.
- **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.*

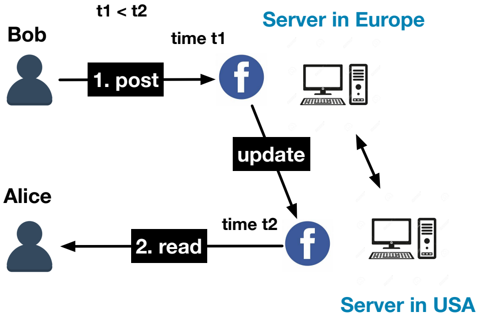
**Proof**

- 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)**.

☞ When there isn't any network partition, the CAP theorem **does not** impose constraints on availability or consistency.

# The CAP theorem

## Why choosing availability over consistency?




Alice does not see Bob's post between $t1$ and $t2$.
**Is it really an issue?**

# CAP theorem and NoSQL databases

## CP Databases

- MongoDB.
- CouchDB.
- Redis.
- HBese.

## AP databases

- Cassandra.
- DynamoDB.

# 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.

# Aggregate

- 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.

| article_id | name | producer | categories |
|---|---|---|---|
| 234543 | Bamboo utensil spoon | KitchenMaster | home, kitchen, spatulas |

☞ In a relational database, we can address this problem by **normalizing** the table.

# Aggregate vs relational row

## 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**.

| article_id | name | producer | categories |
|---|---|---|---|
| 234543 | Bamboo utensil spoon | KitchenMaster | home |
| 234543 | Bamboo utensil spoon | KitchenMaster | kitchen |
| 234543 | Bamboo utensil spoon | KitchenMaster | spatulas |

☞ 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.

| article | | |
|---|---|---|
| **article_id** | **name** | **producer** |
| 234543 | Bamboo utensil spoon | KitchenMaster |

| article_category | |
|---|---|
| **article_id** | **category_id** |
| 234543 | 1 |
| 234543 | 2 |
| 234543 | 3 |

| category | |
|---|---|
| **category_id** | **name** |
| 1 | kitchen |
| 2 | home |
| 3 | spatulas |

# Aggregate vs relational row

## Aggregate

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

```
{
  "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.

```
{
  "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.

```
{
  "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
    }
  ]
}
```

```
{
  "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.

| code_emp | first_name | last_name | salary | position | dept_code | dept_name | budget |
|---|---|---|---|---|---|---|---|
| 234543 | John | Smith | 50000 | Assistant director | 12 | Accounting | 120000 |
| 234543 | John | Smith | 50000 | Assistant director | 145 | HR | 250000 |

☞ The update of the salary of a single employee might be a **cross-node operation**.

```
{
  "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
    }
  ]
}
```

☞ Updating the information on a department is a **non-atomic operation**

```
{
  "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'll see how to alleviate this problem when we introduce MongoDB.

# Aggregate-based NoSQL databases

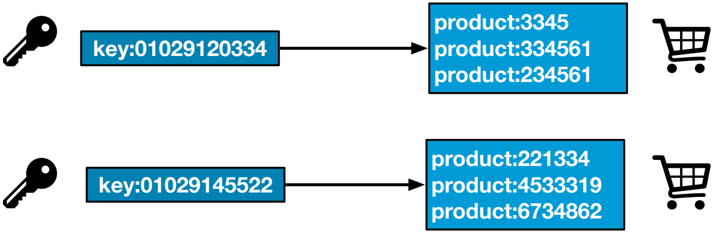
- Aggregates are **schemaless**.
  - No need to adhere to a rigid schema.
  - Flexible evolution of the database.
- Normalization is not required.
  - We accept some **redundancies** in exchange of faster queries.
  - Remember: storage hardware is **cheap** today.
- All data in an aggregate is stored 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**.

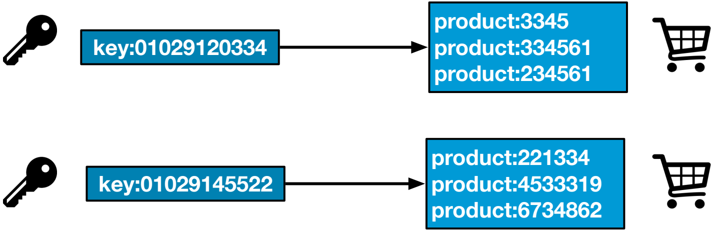
- **Key**: alphanumeric string, usually auto-generated by the database.
- **Value**: an aggregate.
- **Query**: get an aggregate given its key.



# 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.

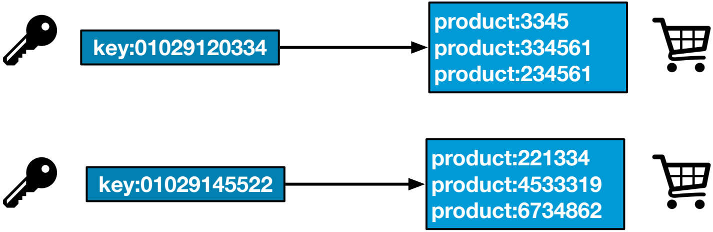


# Key-value databases

## 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.

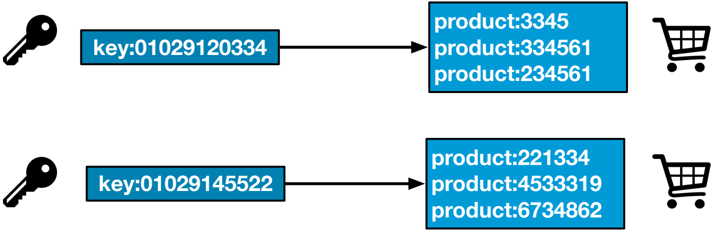


# Key-value databases

## Application scenarios

**Scenario 2. Shopping cart**.

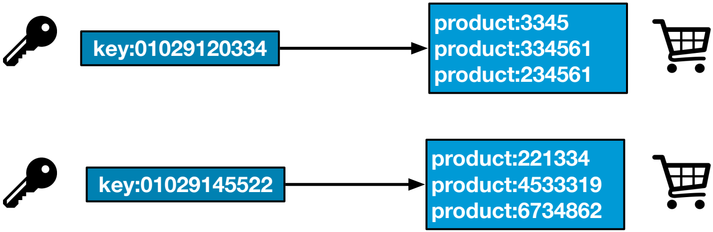
- An e-commerce website may receive billions of orders in seconds.
- Each shopping cart has a **unique identifier** (the key).
- Shopping cart data is only queried by the identifier.
- Shopping cart data can be easily replicated to handle node failures.



# Key-value databases

## Existing key-value databases

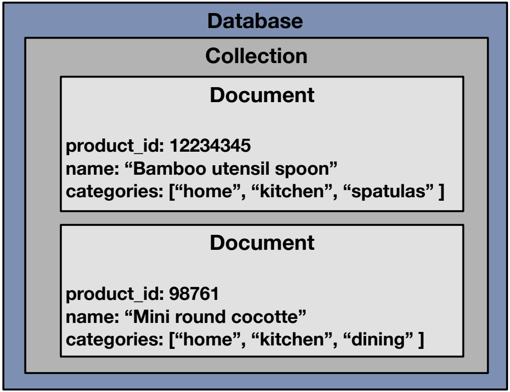
- **Amazon DynamoDB**. One of the first NoSQL databases.
- **Riak**.
- **Redis**. Possibility of tuning data persistence.
- **Voldemort**.



# Document-oriented databases

## Idea

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

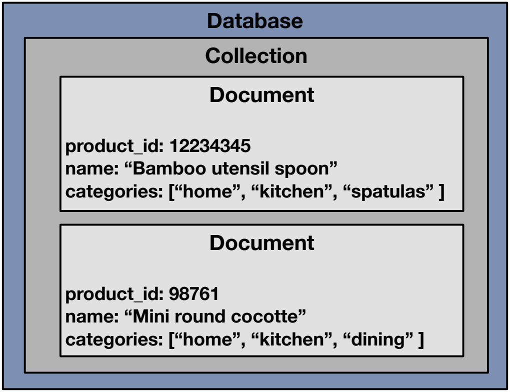



It is possible to search for all products in category *kitchen*.

# Document-oriented databases

## Existing document-oriented databases

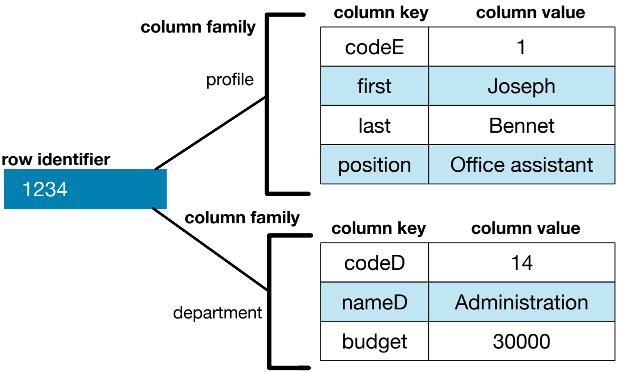
- **MongoDB**, **CouchDB**, **OrientDB**.



# Column-oriented databases

## Idea

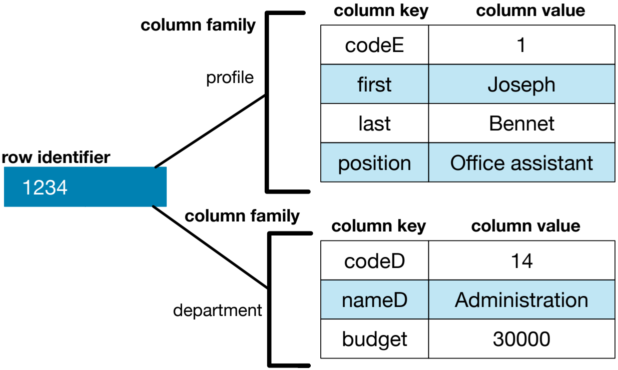
- Similar to document-oriented database but. an aggregate can be broken into smaller data units called **columns**.



# Column-oriented databases

## Idea

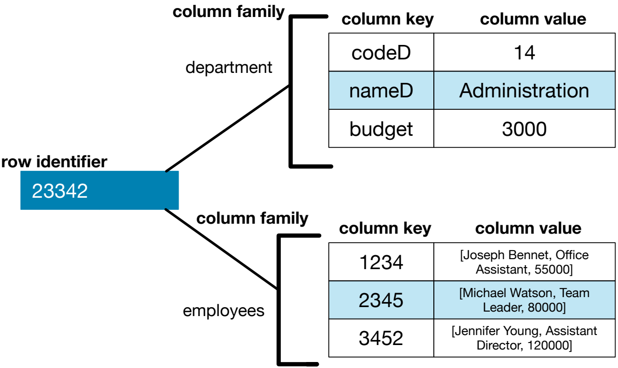
- Columns can be organized into **column families**.
- Columns in the same family are stored on the same node.



# Column-oriented databases

## Idea

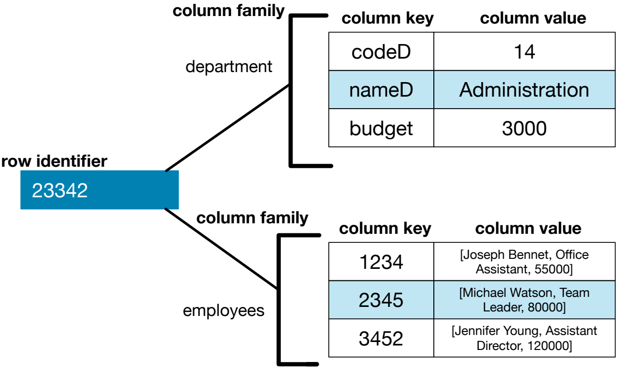
- The value of a column can be an aggregate (**wide column**).



# Column-oriented databases

## Existing column-oriented databases

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



# Graph databases

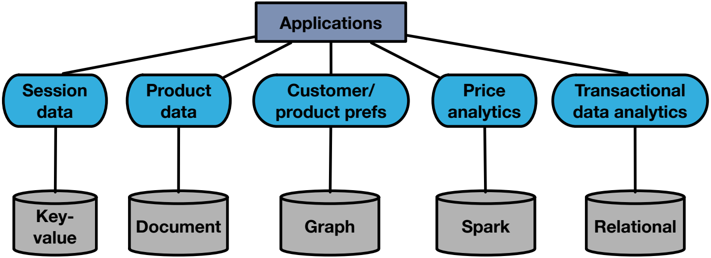
## Idea

- 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**.



# References


- Hoffer, Jeffrey A. *Modern Database Management.* 10/e. Pearson Education India, 2011. Click here