NoSQL

Julien Romero - Télécom SudParis

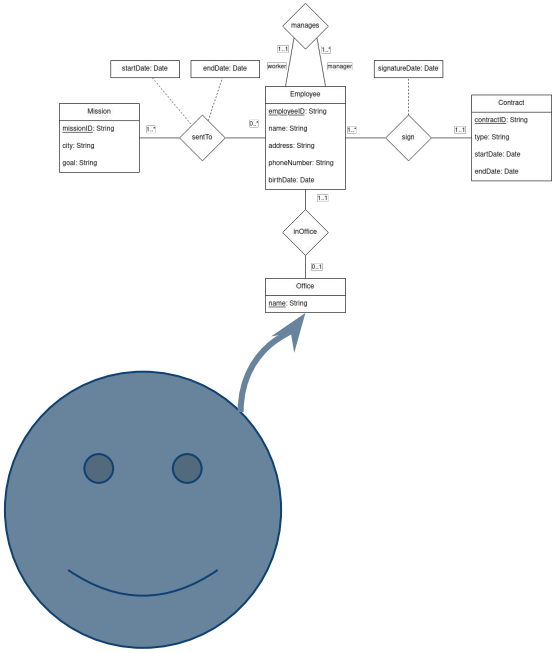
Previously...

The World of Relational Data - Client Needs

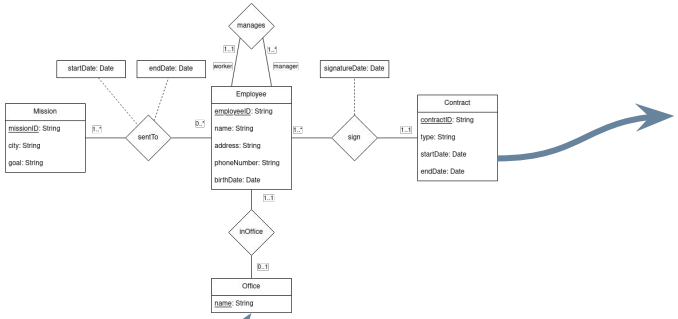
I need an application to manager the employee of my company. Each employee belongs to a department...

Client

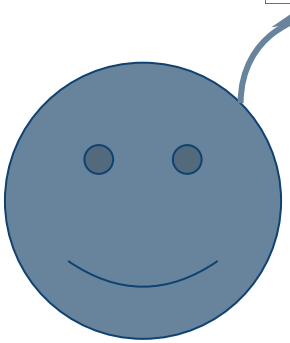
The World of Relational Data - E/R Diagram



The World of Relational Data - Database Schema

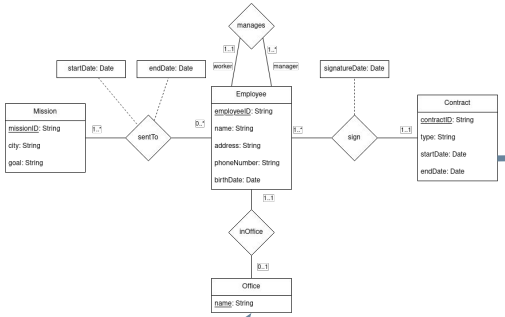


Employees(employeeID: String, name: String, ...)
Mission(missionID: String, city: String, ...)
Contract(contratID: String, type: String, ...)
....



Client

The World of Relational Data - Schema Implementation

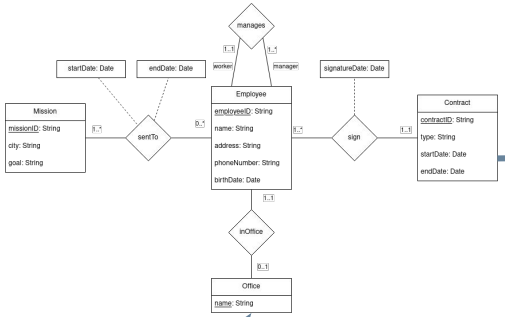


Employees(<u>employeeID</u>: String, name: String, ...)
Mission(<u>missionID</u>: String, city: String, ...)
Contract(<u>contratID</u>: String, type: String, ...)
....

Client

RDBMS

The World of Relational Data - Data Update



Employees(employeeID: String, name: String, ...)
Mission(missionID: String, city: String, ...)
Contract(contratID: String, type: String, ...)
....

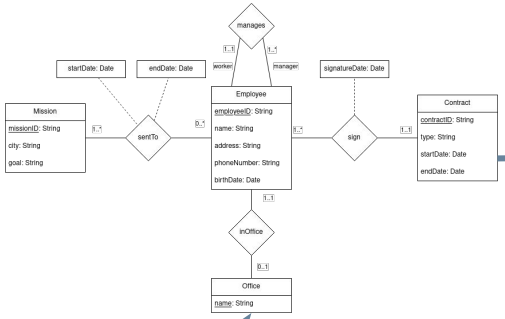
Update

RDBMS

employeeID	name	address	birthdate
E1	Kim	Paris	01/01/95
E3	John	Lyon	10/10/88

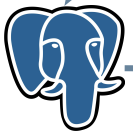
Client

The World of Relational Data - Querying



Employees(employeeID: String, name: String, ...)
Mission(missionID: String, city: String, ...)
Contract(contratID: String, type: String, ...)
....

Client



RDBMS

Update

employeeID	name	address	birthdate
E1	Kim	Paris	01/01/95
E3	John	Lyon	10/10/88

Query

Why Limit Ourselves To Relational Data?

Why Do We Use Relational Databases So Much?

- Intuitive representation, with a mathematical support (relational algebra).
- Very optimized softwares: They have matured over years and are widely used.
- Most RDBMS follow the ACID properties...

ACID

General Problems - Atomicity

An operation (also called **transaction**) on a database (like read and write) is often composed of many sub-operations. For example:

- Give me the list of my friends who live in Paris =
  - Get my list of friends
  - Get the address of all my friends
  - Keep only the ones who live in Paris

The **atomicity** ensure that my transaction is treated as a single “unit” that either succeeds entirely or fails entirely.

General Problems - Atomicity

Send 100€ on my friend =

- Check if I have 100€ on the first account
- If so, remove 100€ on the first account
- Add 100€ to the account of my friend.

What if the third operation fails? (e.g. my friend gave me a wrong account ID or the servers shutdown)

Atomicity guarantees that the entire transaction fails: No money was actually withdrawn from my account.

General Problems - Consistency

The state of the database before and after a transaction remains consistent, i.e. it respects some **integrity constraints**.

Example: Internal operations in a bank do not create or destroy money.

In a bank, we do not want to allow queries that remove 100€ from an account and add 200€ in another account.

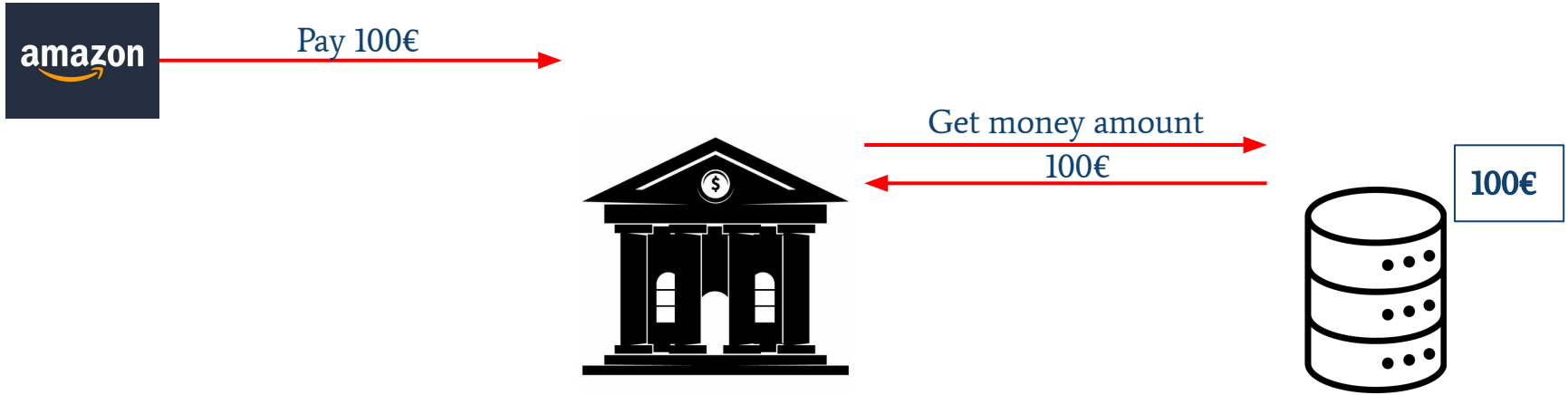
General Problems - Isolation

If two transactions happen at the same time, it is like they happen sequentially.

We might have a lot of troubles if many transactions try to access/write the same data at the same time.

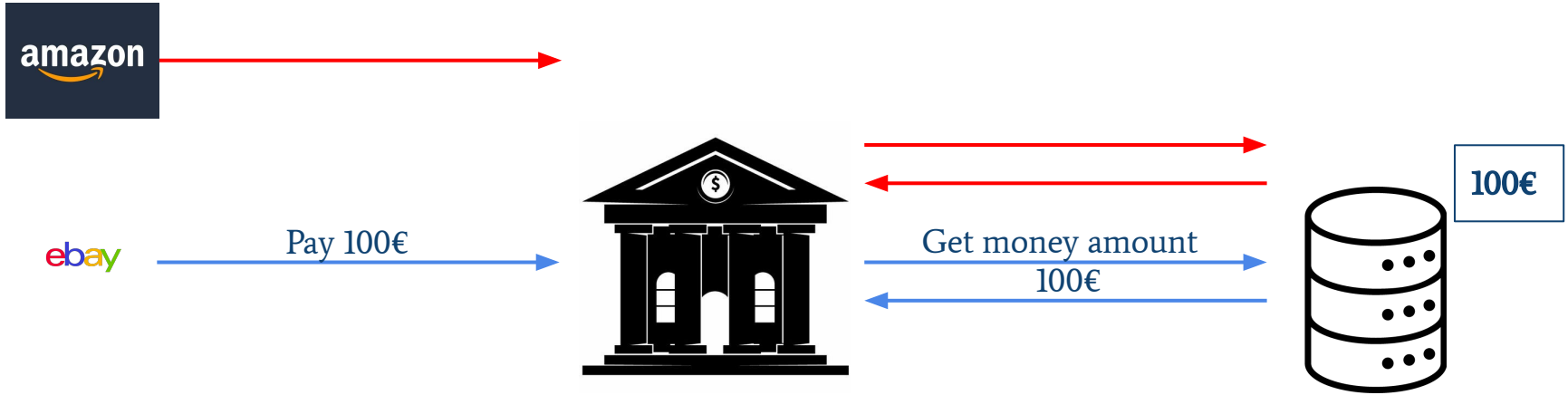
General Problems - Isolation - Example

Pay 100€ from my account = Read the total amount of money + set the new value if possible



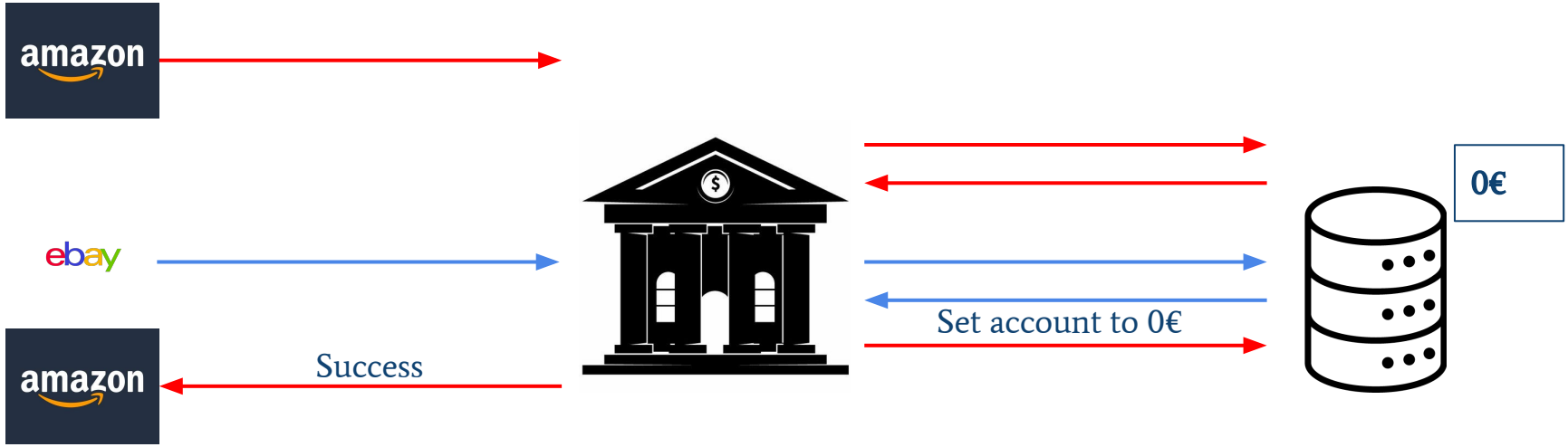
General Problems - Isolation - Example

Pay 100€ from my account = Read the total amount of money + set the new value if possible



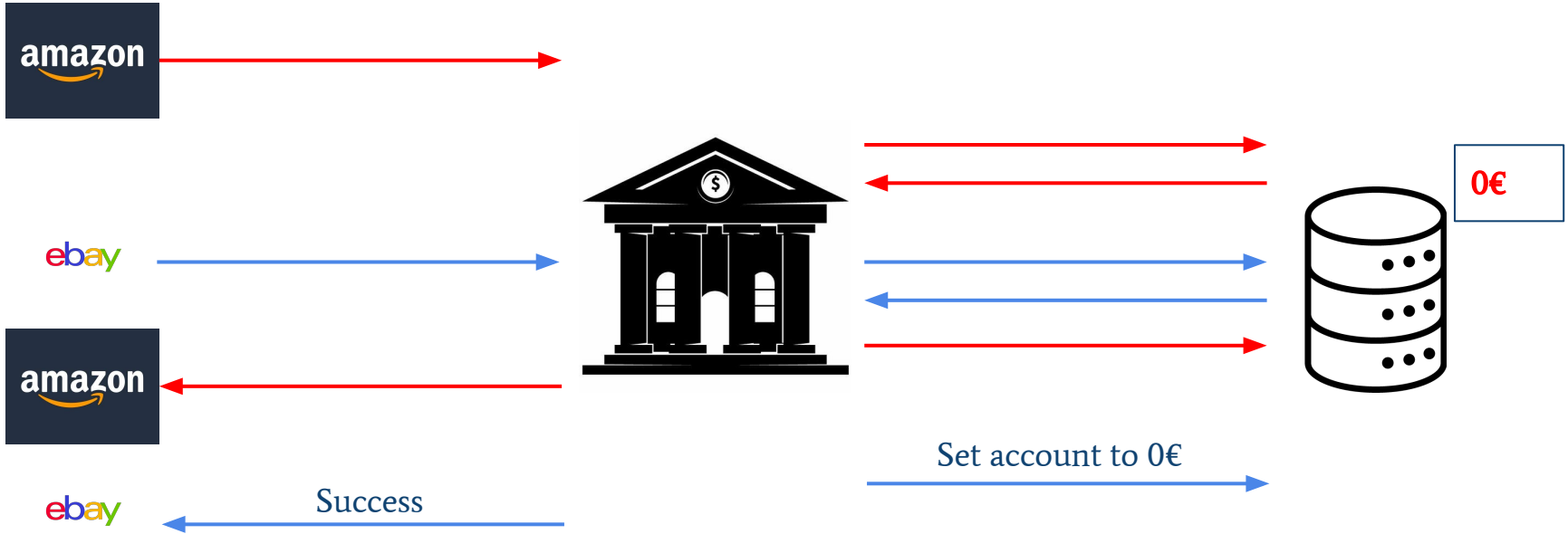
General Problems - Isolation - Example

Pay 100€ from my account = Read the total amount of money + set the new value if possible



General Problems - Isolation - Example

Pay 100€ from my account = Read the total amount of money + set the new value if possible



General Problems - Durability

If a transaction is successful, its effects are permanent even if the system fails.

Example: I send 100€ to my friend and the operation is successful. Even if the servers of the bank crash, I will still have 100€ less and my friend 100€ more after the servers are back.

General Problems - ACID

ACID (Atomicity, Consistency, Integrity, Durability) is a set of properties that guarantee the quality of the database in case of errors, power failures or other kinds of mishaps.

Limitations of ACID

- ACID has a cost = Systems are slower.
- Poor scalability
  - Hard to parallelize.
  - Big data companies have to use several computers to collect their data.
  - ACID can be VERY expensive on several computers

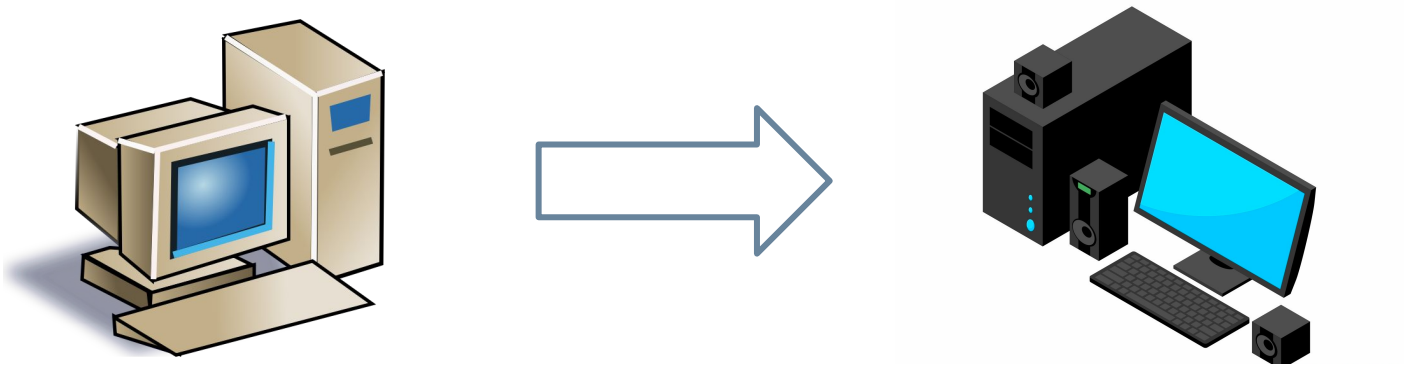
What is Scalability?

"Scalability is the property of a system to handle a **growing amount of work** by **adding resources to the system.**" [Wikipedia]

It means that your application can grow smoothly and adapt to the usage.

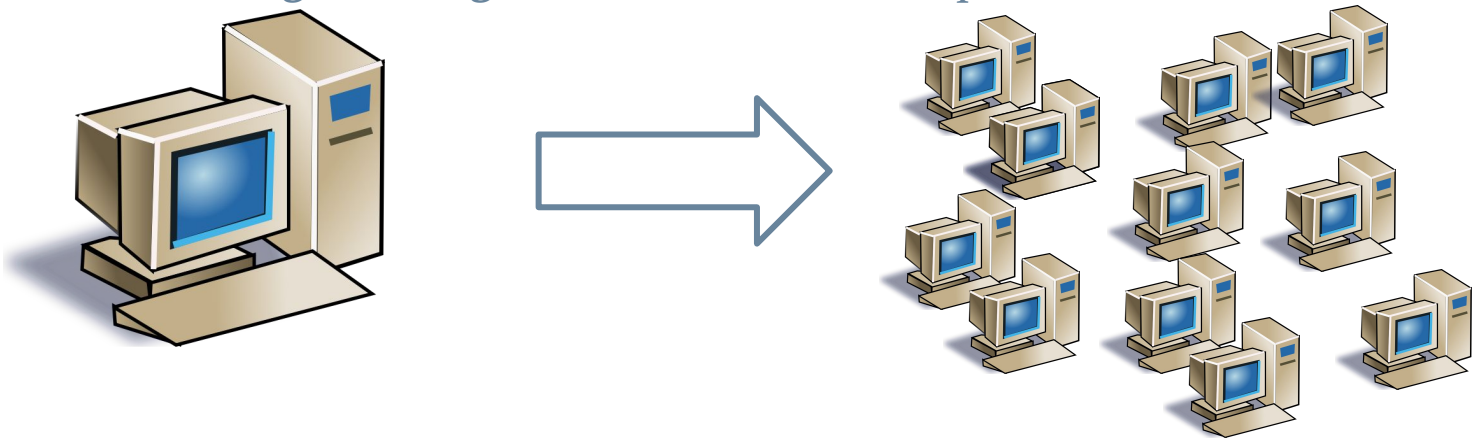
How To Scale?

- Horizontal vs vertical scaling
- Vertical scaling (“scaling up”): Improve the computers you currently have.
  - More processing power
  - More memory



How To Scale?

- Horizontal vs vertical scaling
- Vertical scaling ("scaling up"): Improve the computers you currently have.
  - More processing power
  - More memory
- Horizontal scaling ("scaling out"): Add more (cheap) machines



Why Is Scaling Out Better?

- Adding resources on a single computer is harder and harder, and more and expensive.
- Resources on a single machine are limited. By scaling out, I can add as many machines as I want.
- If your single computer dies, all your system dies if you have a single machine.
- Changing the single computer takes time. With many computers, you can have a smooth transition
- ...

Take a lecture about the cloud to learn more!

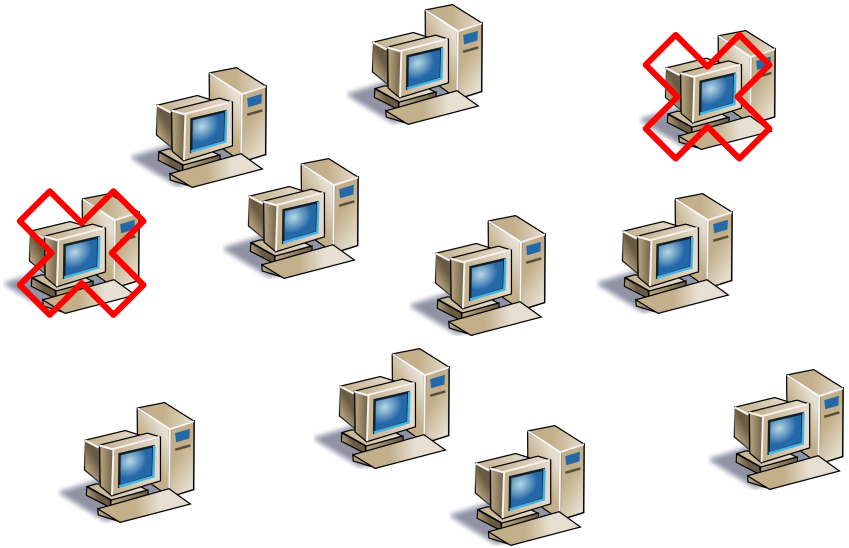
NoSQL

What is NoSQL?

- A NoSQL database is a database that does not follow the relational model
- NoSQL does not mean Not SQL
  - NoSQL is a database kind, not a programming language
  - NoSQL databases often allow the usage of SQL
  - NoSQL = Not Only SQL
- In general, a NoSQL database is:
  - Non-relational: Not only tables
  - Distributed: Can be on several machines, all around the world
  - Scalable: Store and query large amount of data
  - Available: Even if a machines crashes, continues working

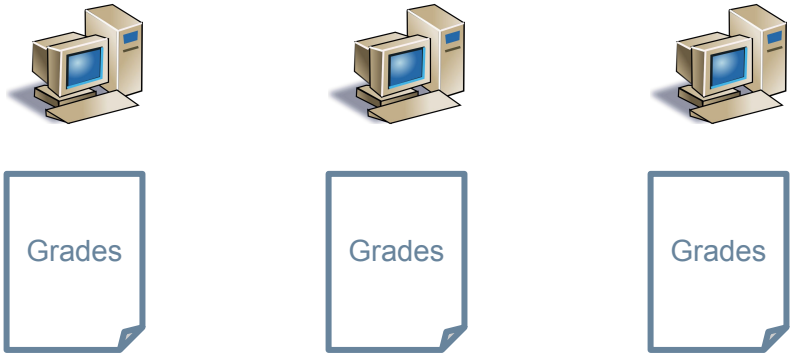
Availability And Duplication

- We have data stored on several machines. How can we protect the data if one or several machines crash?



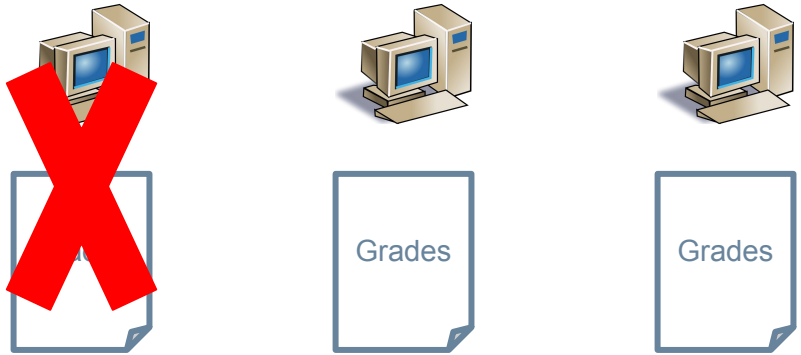
Availability And Duplication

- We have data stored on several machines. How can we protect the data if one or several machines crash?
- We use duplication!
  - Instead of storing the data only once, we store it several times.



Availability And Duplication

- We have data stored on several machines. How can we protect the data if one or several machines crash?
- We use duplication!
  - Instead of storing the data only once, we store it several times.



Even if a machine crashes, the data is still available

Why Do We Want To Use NoSQL?

- Application development productivity
  - Organizing relational data can take a lot of time
  - NoSQL is less structured, allowing for flexibility, easier changes, and faster prototyping.
- Large Data Scale
  - Storing large amount of data using a relational database is expensive.
  - Typically less expensive to have a database on several small machines rather than a sing big one.

NoSQL vs SQL

NoSQL Database

- Uses SQL, or not
- Not only tables
- Flexible schema = can change
- Scales out = can add more machines

Relational

- Uses SQL
- Tables with predefined columns and rows
- Fixed schema = hard to change
- Scales up = need more powerful machines

The Four Types of NoSQL Databases

- Tabular
- Key-Value
- Document
- Graph

Tabular Data

Very Similar To Relation Model

- Many NoSQL databases allow to have tables with columns and rows
- Very similar to the relation model we saw in previous lectures.

Key-Value Data

Key-Value Data

- Very simple. Basically a table with two columns:
  - A unique key
  - A value associated to each key
- Example:

Key	Value
France	Paris
Germany	Berlin
Spain	Madrid

Key-Value Data - Advantages

- Simplicity: Almost no structure and type constraints.
- Speed: Very fast
  - Very good for cache

Key-Value Data - Disadvantages

- Cannot search by value
  - I cannot find what is the country with Paris as a capital
- Cannot easily associate several values to a single key
  - StudentID and multiple grades
- Cannot modify partially the value, we have to modify everything

Document Data

The JSON Format

JSON is a very popular file type that structures information. A JSON document can be:

- A value (String, integer, float).
  - E.g: 1, “Paris”, 3.14
- A list of values.
  - E.g.: [1, 2, “France”, 99.2]
- An association of keys and values.
  - Eg. {“France”: “Paris”, “Germany”: “Berlin”, “Spain”: “Madrid”}
- Values can also be JSON documents !
  - We can compose the JSON documents.

The JSON Format - Example

```
group = {
  “groupname”: “Beatles”,
  “members”: [
     {“firstname”: “John”, “lastname”: “Lennon”, “age”: 42,
      “instrument”: [“guitare”, “voice”]},
     {“firstname”: “Paul”, “lastname”: “Mccartney”, “age”: 75,
      “instrument”: [“guitare”, “voice”]}
    ]
  “albums”: [
     {“name”: “Sgt. Pepper's Lonely Hearts Club Band”, “year”: 1967},
     {“name”: “Yellow Submarine”, “year”: 1969}
    ]
}
```

The JSON Format - Getting data

- To get a partial data in a JSON file, we use the notation myJSON[i] where i is the index of the element in the list (we traditionally start at 0), or the key for an association.
  - With names=[“Paul”, “Jack”, “Alice”, Roxanne”], names[2] is “Alice”
  - With student={“name”: “John”, “age”: 18, “studentID”: “S12”}, student[“name”] is “John”
- As a JSON document can also contain a JSON document, we can chain the partial accesses.
  - myJSON[2][“student”][15][1]

The JSON Format - Example

```
group = {
  “groupname”: “Beatles”,
  “members”: [
     {“firstname”: “John”, “lastname”: “Lennon”, “age”: 42,
      “instrument”: [“guitare”, “voice”]},
     {“firstname”: “Paul”, “lastname”: “Mccartney”, “age”: 75,
      “instrument”: [“guitare”, “voice”]}
    ]
  “albums”: [
     {“name”: “Sgt. Pepper's Lonely Hearts Club Band”, “year”: 1967},
     {“name”: “Yellow Submarine”, “year”: 1969}
    ]
}
```

```
group[“members”][1][“instrument”][0]
```

The JSON Format - Example

```
group = {
  “groupname”: “Beatles”,
  “members”: [
     {“firstname”: “John”, “lastname”: “Lennon”, “age”: 42,
      “instrument”: [“guitare”, “voice”]},
     {“firstname”: “Paul”, “lastname”: “Mccartney”, “age”: 75,
      “instrument”: [“guitare”, “voice”]}
    ]
  “albums”: [
     {“name”: “Sgt. Pepper's Lonely Hearts Club Band”, “year”: 1967},
     {“name”: “Yellow Submarine”, “year”: 1969}
    ]
}
```

```
group[“members”][1][“instrument”][0]
```

The JSON Format - Example

```
group = {
  “groupname”: “Beatles”,
  “members”: [
     {“firstname”: “John”, “lastname”: “Lennon”, “age”: 42,
      “instrument”: [“guitare”, “voice”]},
     {“firstname”: “Paul”, “lastname”: “Mccartney”, “age”: 75,
      “instrument”: [“guitare”, “voice”]}
    ]
  “albums”: [
     {“name”: “Sgt. Pepper's Lonely Hearts Club Band”, “year”: 1967},
     {“name”: “Yellow Submarine”, “year”: 1969}
    ]
}
```

group[“members”][1][“instrument”][0]

The JSON Format - Example

```
group = {
  “groupname”: “Beatles”,
  “members”: [
     {“firstname”: “John”, “lastname”: “Lennon”, “age”: 42,
      “instrument”: [“guitare”, “voice”]},
     {“firstname”: “Paul”, “lastname”: “Mccartney”, “age”: 75,
      “instrument”: [“guitare”, “voice”]}
    ]
  “albums”: [
     {“name”: “Sgt. Pepper's Lonely Hearts Club Band”, “year”: 1967},
     {“name”: “Yellow Submarine”, “year”: 1969}
    ]
}
```

group[“members”][1][“instrument”][0]

The JSON Format - Example

```
group = {
  “groupname”: “Beatles”,
  “members”: [
     {“firstname”: “John”, “lastname”: “Lennon”, “age”: 42,
      “instrument”: [“guitare”, “voice”]},
     {“firstname”: “Paul”, “lastname”: “Mccartney”, “age”: 75,
      “instrument”: [“guitare”, “voice”]}
    ]
  “albums”: [
     {“name”: “Sgt. Pepper's Lonely Hearts Club Band”, “year”: 1967},
     {“name”: “Yellow Submarine”, “year”: 1969}
    ]
}
```

group[“members”][1][“instrument”][0]

Document Database

- A document database stores documents in a predefined format like JSON.
- The documents do not have to follow any structure.

Document Database - Advantages

- No schema: The documents can take any form. This is good for changing applications.
- Easy to update: We can update parts of a document.
- Fast as a document does not rely on additional information like in relational tables.

Document Database - Disadvantages

- Hard to check consistency: Because documents are independent, they might carry similar information, sometimes inconsistent.
- Atomicity issue: Cannot modify two documents in a single transaction.

Turning A Relation Database Into A Document Database

We want to turn the following database into one document for each wine:

- Wines(WineID: String, vineyard: String, year: Integer, degree: Float)
- Harvests(WineID: String, ProducerID: String, weight: Float)
- Producers(ProducerID: String, name: String, city: String)
- Clients(ClientID: String, name: String, city: String)
- Orders(orderID: String, date: Date, ClientID: String, WineID: String, quantity: Float)

Turning A Relation Database Into A Document Database

1. Create the fields for the main relation schema.

- Wines(WineID: String, vineyard: String, year: Integer, degree: Float)

```
wine = {
  "wineID": "W12",
  "vineyard": "Chinon",
  "year": 2015,
  "degree": 13.5,
}
```

Turning A Relation Database Into A Document Database

2. For each relation involving a wine, add a new field. The new value will be a list where each element is a row matching the wineID.

(you do not need to repeat the wineID).

- Harvests(WineID: String, ProducerID: String, weight: Float)

```
wine = {
  "wineID": "W12",
  "vineyard": "Chinon",
  "year": 2015,
  "degree": 13.5,
  “harvests”: [{“producerID”: “P159”, “weight”: 17}, {“producerID”: “P789”, “weight”: 98}]
}
```

Turning A Relation Database Into A Document Database

3. If you key primary keys, we can decide to replace it by the full row associated with this key.

Note: this creates a lot of redundancy!

- Producers(ProducerID: String, name: String, city: String)

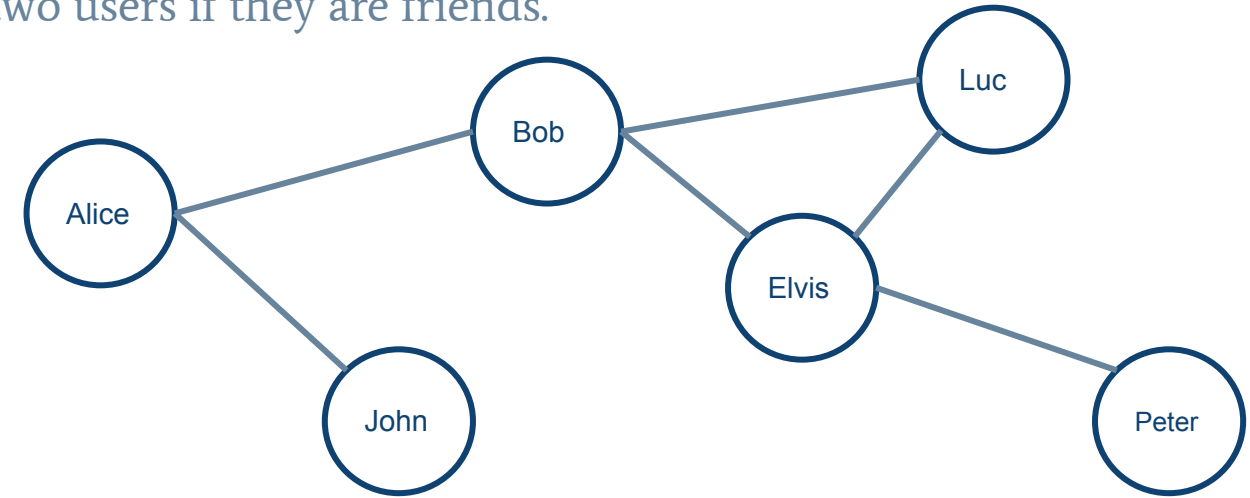
```
wine = {
  "wineID": "W12",
  "vineyard": "Chinon",
  "year": 2015,
  "degree": 13.5,
  “harvests”: [
     {“producer”:
         {“producerID”:  “P159”, “name”: “Luke Soin”, “city”: “Bordeaux”},
      “weight”: 17},
     ...]
}
```

Graph Data

What Is a Graph?

A graph is a set of nodes representing entities that are connected with edges.

Example: The friendship graph of Facebook. The nodes are the users, and there is a link between two users if they are friends.



Graph Databases

- We can store graphs into graph databases
- We can exploit the structure of the graph with the queries
  - E.g.: Find a path between two nodes, count the number of friends.
- We can also attach properties to nodes and edges.
  - E.g.: name, birthdate to a node representing a friendship
  - E.g.: The start of the relationship for an edge

Graph Databases - Advantages

- Flexible structure
- Easy to understand

Graph Databases - Disadvantages

- Not all data can be easily expressed with a graph.
- Longer to query than tables in some cases.

Turning A Relation Database Into A Document Database

When we want to turn a relational database into a graph database, we have to think in terms of E/R diagrams. The entities are the nodes, and the relationships are edges. The additional properties can be attached to the nodes and edges (not seen in this lecture).

- Pilots(pilotID: String, name: String, birthdate: Date)
- Planes(planeID: String, buildDate: Date, numberOfSeats: Integer)
- usePlane(flightID: String, planeID: String)
- hasPilot(flightID: String, pilotID: String)
- departureAirportFlight(flightID: String, airportID: String, gate: String)
- Airports(airportID: String, name: String, city: String)
- canPilot(pilotID: String, planeID: String)

Turning A Relation Database Into A Document Database

1. Identify the entities.

- Pilots(pilotID: String, name: String, birthdate: Date)
- Planes(planeID: String, buildDate: Date, numberOfSeats: Integer)
- usePlane(flightID: String, planeID: String)
- hasPilot(flightID: String, pilotID: String)
- departureAirportFlight(flightID: String, airportID: String, gate: String)
- Airports(airportID: String, name: String, city: String)
- canPilot(pilotID: String, planeID: String)

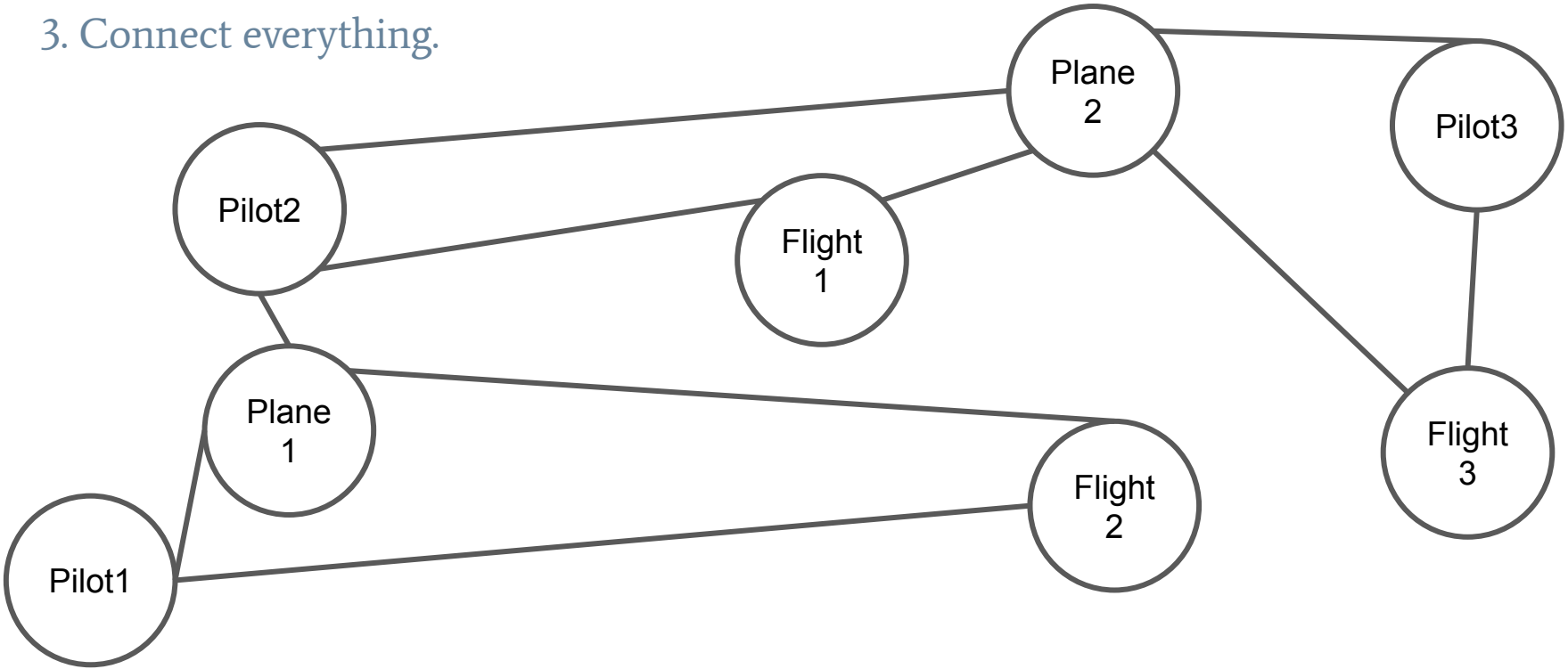
Turning A Relation Database Into A Document Database

2. Identify the relationships.

- Pilots(pilotID: String, name: String, birthdate: Date)
- Planes(planeID: String, buildDate: Date, numberOfSeats: Integer)
- usePlane(flightID: String, planeID: String)
- hasPilot(flightID: String, pilotID: String)
- departureAirportFlight(flightID: String, airportID: String, gate: String)
- Airports(airportID: String, name: String, city: String)
- canPilot(pilotID: String, planeID: String)

Turning A Relation Database Into A Document Database

3. Connect everything.



Summary

Summary

- Most RDBMS follow ACID.
- The relational model is very rigid.
- NoSQL (=Not Only SQL) brings flexibility, allows data distribution, and is more scalable.
- Four types of NoSQL databases:
  - Tabular
  - Key-Value
  - Document
  - Graph