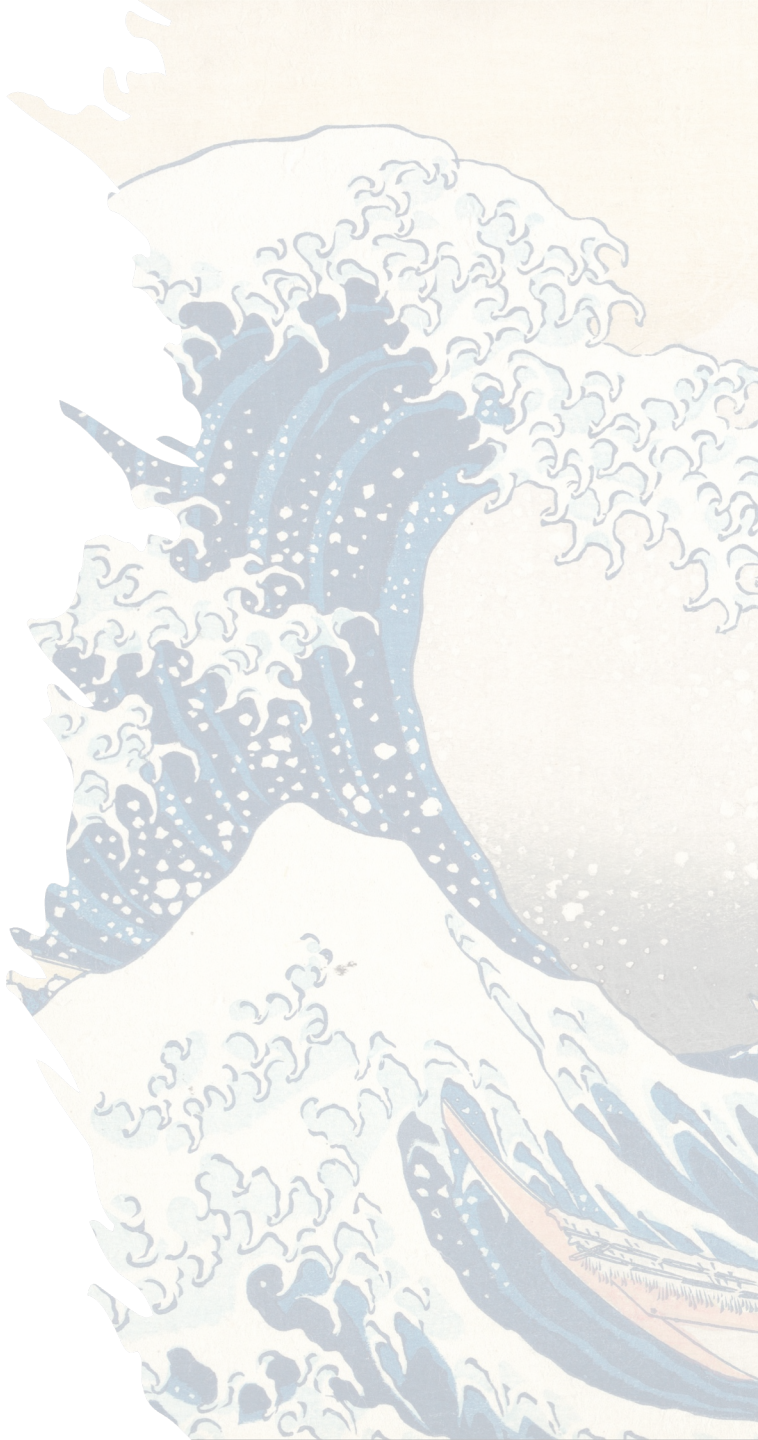
Stelios Sotiriadis

7. Distributed databases for Big Data

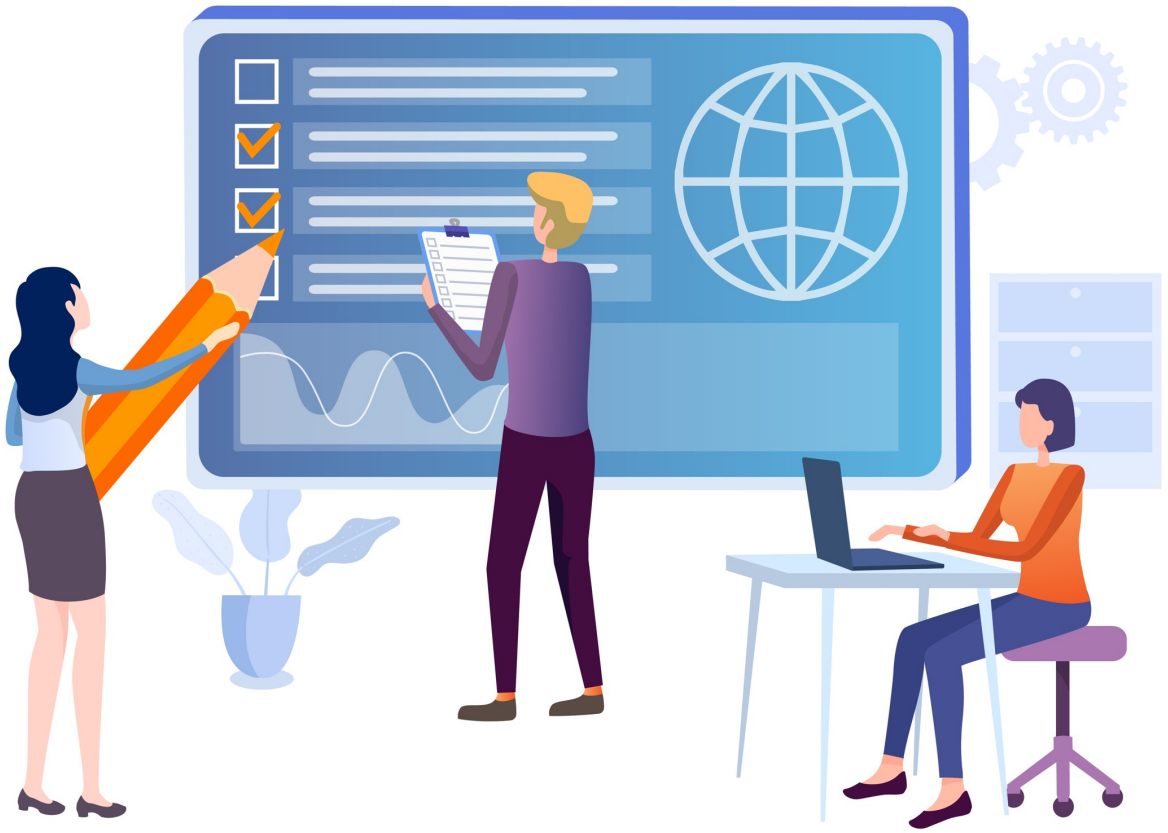
Agenda

- Properties of databases
  - ACID/CAP/BASE
  - Relational vs NoSQL
- Apache Cassandra Architecture
- Setup a cluster
  - Install an Apache Cassandra cluster of 3 nodes on GCP
  - Develop a Python app to interact with the cluster



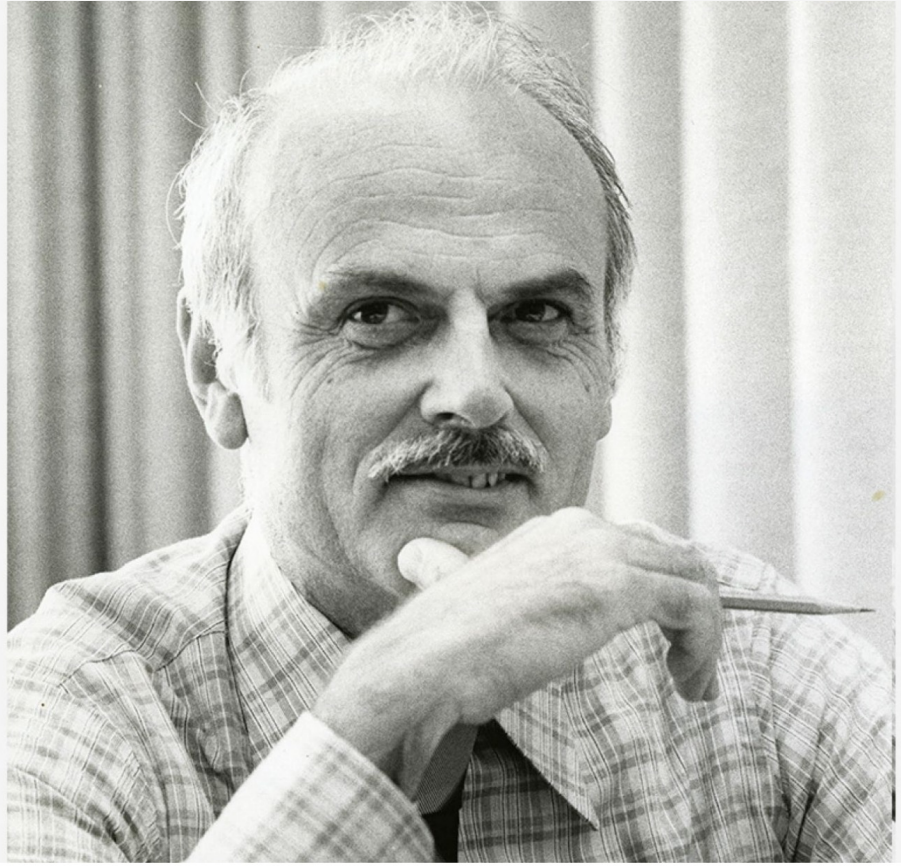
Quiz of the day

Get ready!



Distributed Database Systems

Edgar F. Codd



The inventor made relational databases possible

Edgar F. “Ted” Codd was a mathematician and computer scientist best known for his trailblazing work on the relational model that led to the multibillion-dollar database industry. The revolutionary power of relational databases is taken for granted today, but in 1970 the concept was merely theoretical.

That’s when Codd, an Oxford-educated mathematician working at the IBM San Jose Research Laboratory (now IBM Research – Almaden) in San Jose, California, published a paper describing a system that could store and access information without providing a formal organizational structure or even recording exact locations for data. Until that time, retrieving information typically involved employing specialists to write programs to extract precise data lots. Relational databases shifted this dynamic, making it easy for laypeople to easily access data.

Databases have since become the standard means for accessing everything from bank and credit card accounts to travel reservations, online purchases and stocks.

Database systems

- Relational Database Management System (RDBMS)
  - Basic data model
    - Relations, tuples, attributes and domains
    - Primary and foreign keys
  - Query model:
    - Relational Algebra: Joins!
  - Structure Query Language (SQL)
    - Developed at IBM San Jose Research Laboratory (1970s)

RDBMS: An example data model

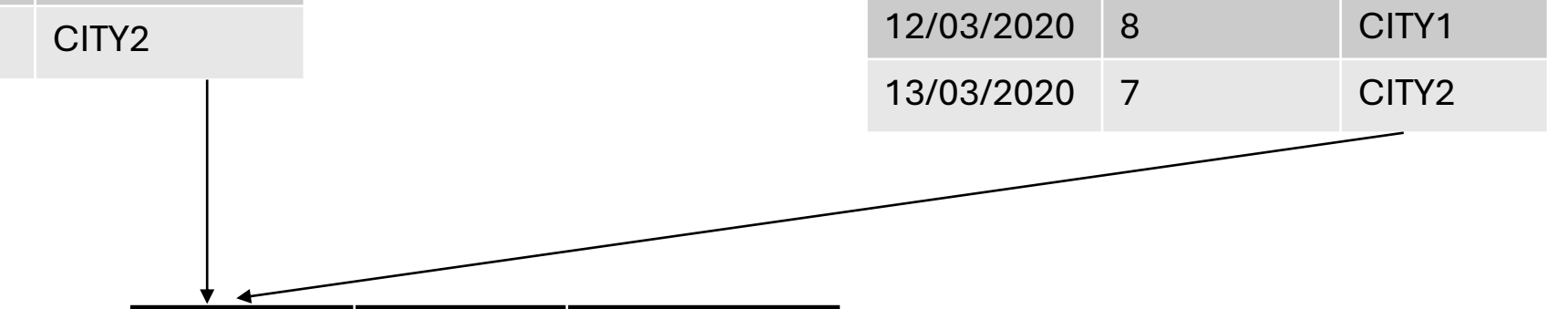
We need to store information about persons living in a city.

PERSON

ID	NAME	CITY_ID
CHAR1	LUKE	CITY1
CHAR2	MARY	CITY2

TEMP

DAY	MIN_TEMP	CITY_ID
12/03/2020	8	CITY1
13/03/2020	7	CITY2



CITY

CITY_ID	CNAME	POPULATION
CITY1	TORONTO	2.93M
CITY2	LONDON	9M

RDBMSs

- A selection of tables in a database that can easily be updated
- **What is Data Redundancy?**
  - It uses keys and indexes in the table to avoid any data redundancies
  - Reduced updating errors and increased consistency

Transactions on RDBMS

- Client transaction with a bank (ATM or browser)
  - Transfer $ 100 from a savings account to a current account
  - Transfer $ 200 from a fixed account to a current account
  - Withdrawal of $ 400 from current and fixed accounts, one after the other

Example of a set of transactions…

- **Transactions**

1. `savings.withdraw(100)`
2. `current.deposit(100)`
3. `fixed.withdraw(200)`
4. `current.deposit(200)`
5. `current.withdraw(400)`
6. `fixed.withdraw(400)`

Failure at this point, we don't lose money…

But we cannot repeat previous steps…

Failure at this point and the customer will lose money…

We need to recover the previous step

ACID: four key properties of a transaction

- **A**tomicity:
  - All or nothing: either all transactions succeed, or all fail
- **C**onsistency:
  - Data is in a consistent state when a transaction starts and when it ends.
- **I**solation:
  - The intermediate state of a transaction is invisible to other transactions
- **D**urability:
  - If a transaction succeeds, its effect is permanent and it survives even if the system fails

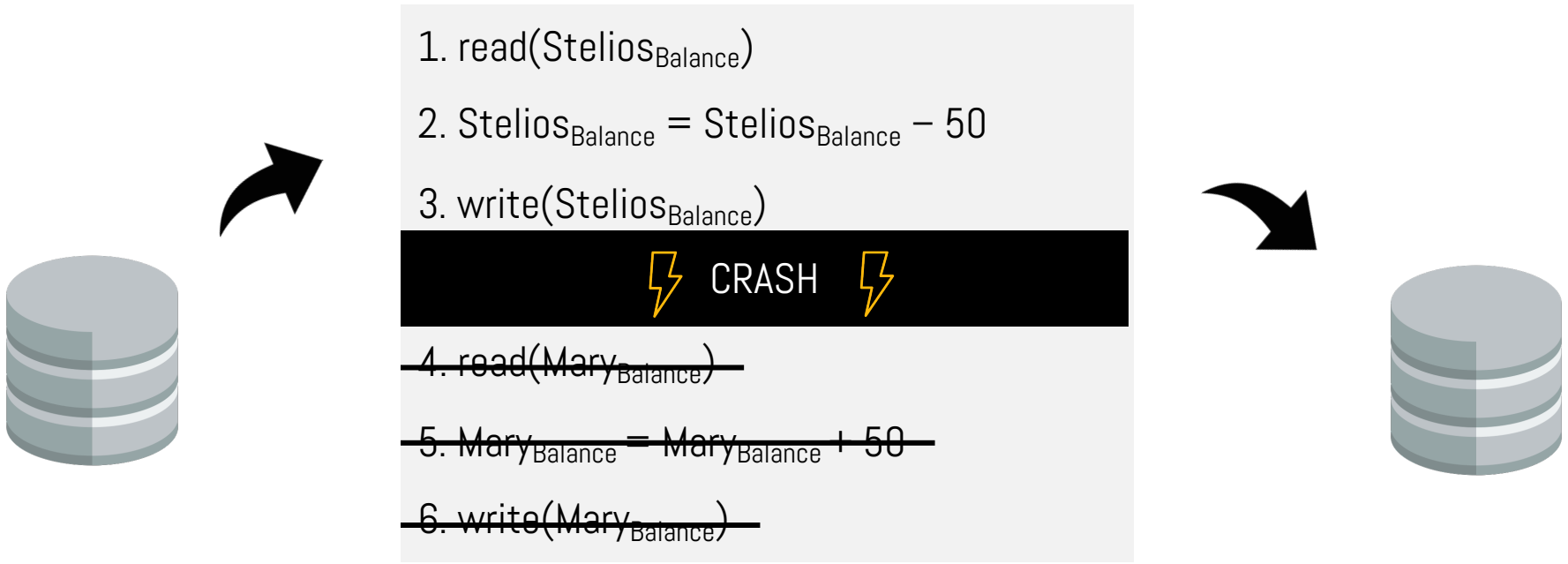
Let us see an example

- Transaction: Transfer of £50 from Stelios to Mary

1. $\mathbf{read(Stelios_{Balance})}$
2. $\mathbf{Stelios_{Balance} = Stelios_{Balance} - 50}$
3. $\mathbf{write(Stelios_{Balance})}$
4. $\mathbf{read(Mary_{Balance})}$
5. $\mathbf{Mary_{Balance} = Mary_{Balance} + 50}$
6. $\mathbf{write(Mary_{Balance})}$

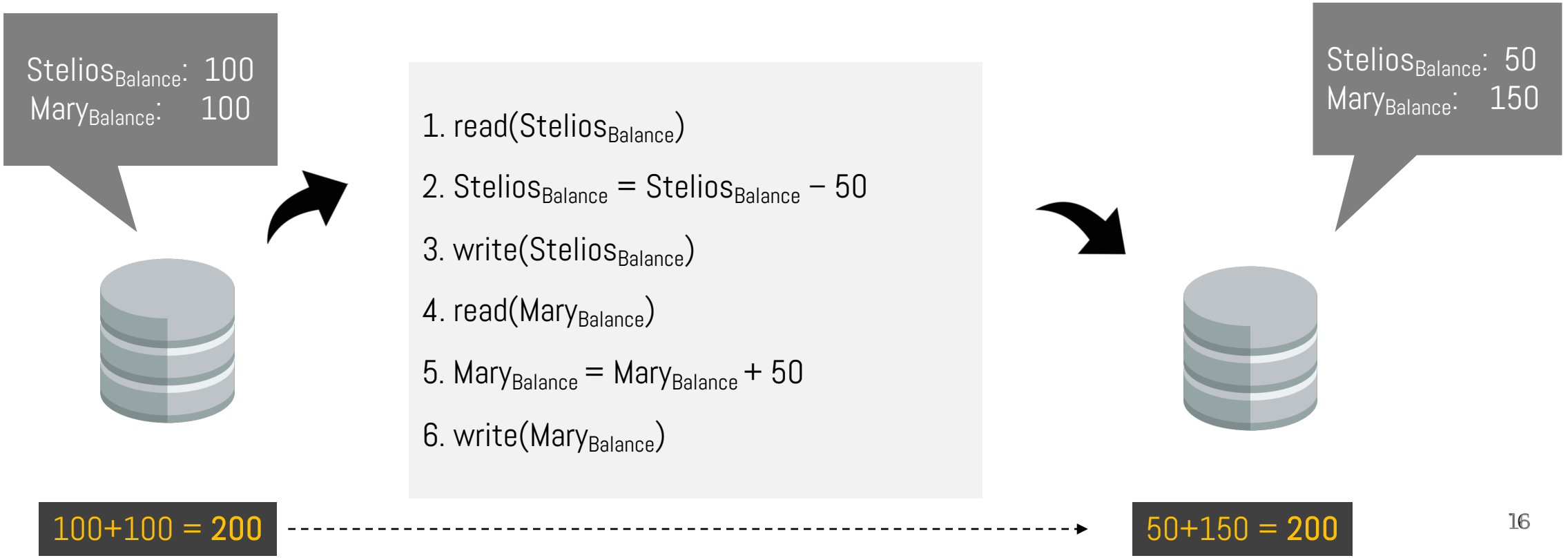
Atomicity: All or nothing!

- If the transaction fails after step 3 and before step 6, the system must ensure steps 1-3 are undone!
  - Otherwise, we will have a partially successful transaction…



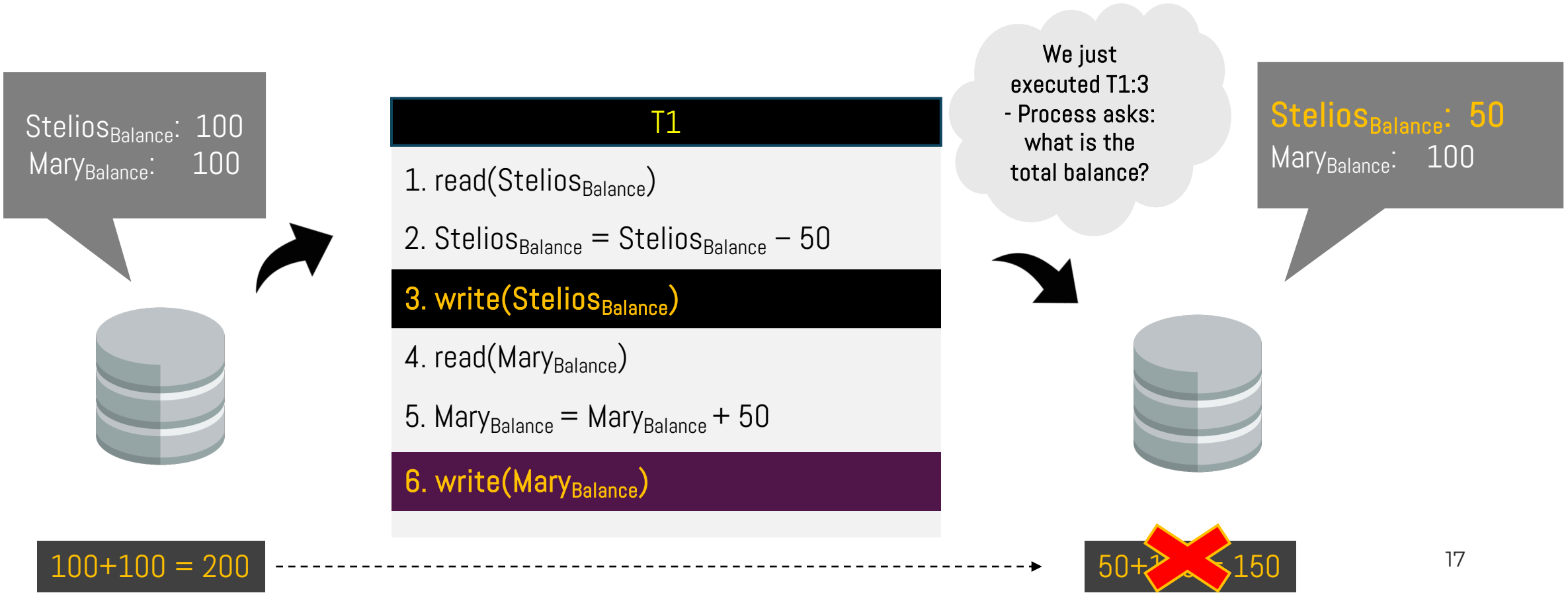
Consistency

- Balance sum remains unchanged at the end of transaction.

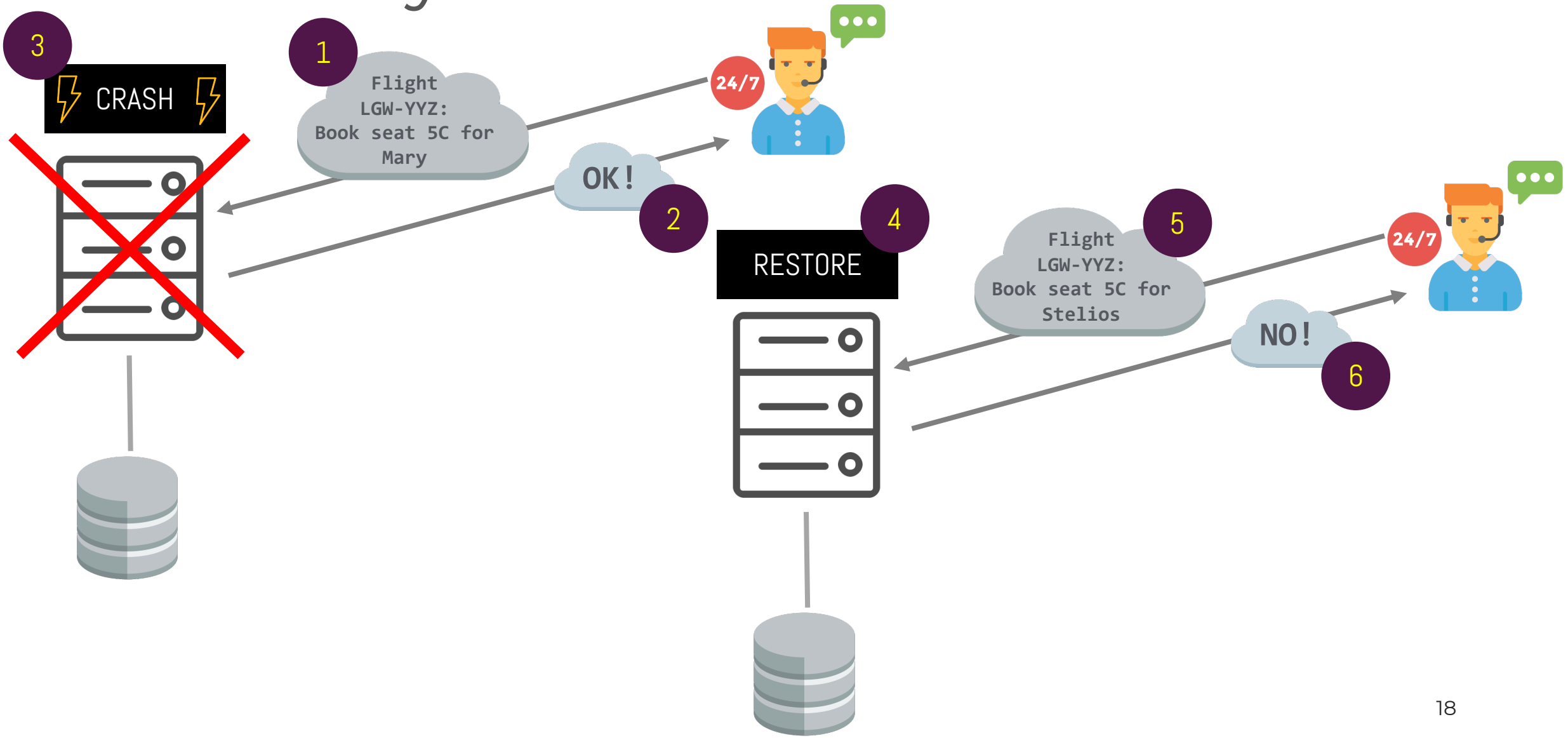


Isolation

- If the system runs more than one transaction at a time and one can see the intermediate results of the other (e.g. , between steps 3 and 6), we may be inconsistent.



Durability


3
CRASH
1
Flight LGW-YYZ: Book seat 5C for Mary
24/7
OK!
2
4
RESTORE
5
Flight LGW-YYZ: Book seat 5C for Stelios
24/7
NO!
6

True or False?

1. Atomicity means that one transaction cannot read data from another.

   False, that's isolation

2. Isolation turns individual operations into an all-or-nothing unit of work, succeeding if and only if all contained operations succeed.

   False, that's atomicity

3. Durability is about maintaining data integrity constraints.

   False, that's consistency

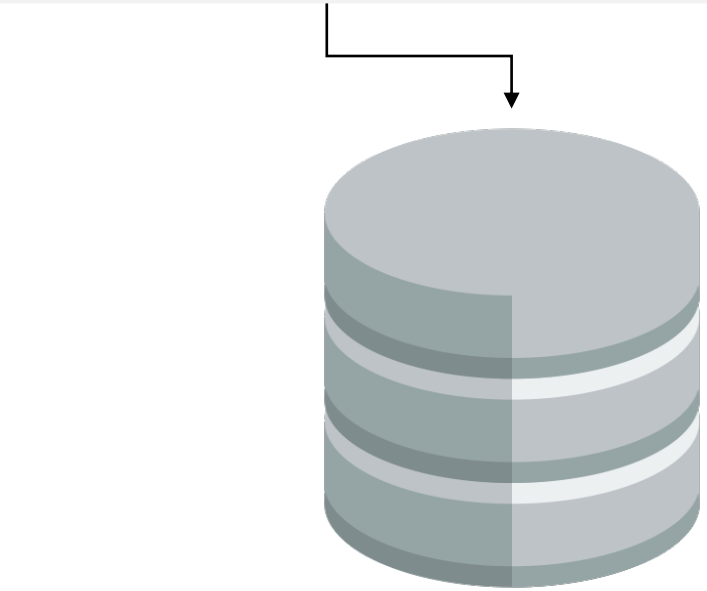
4. If a system is suddenly affected by a system crash or a power outage, all unfinished committed transactions may be replayed.

   False, that's durability

RDBMSs Advantages

```
SELECT CITY.NAME, CITY. POPULATION, PERSON.NAME
FROM CITY, PERSON
WHERE CITY.NAME = 'LONDON'
AND CITY.CITY_ID = PERSON.CITY_ID
```

- Reduced data storage footprint:
  - Removes redundant data
  - The ability to identify and **utilize relationships** inherent in the datasets.
- Use of SQL to query data
  - Easy way of joining data belonging to different tables.



C_NAME	POPULATION	P_NAME
LONDON	9.000.000	MARY

RDBMS disadvantages

- Becomes complex as the amount of data grows
- Technically:
  - Tend to be slow and not scalable (when more storage is needed)
  - Have a fixed schema, so it is quite inflexible in terms of adapting requirements
  - Tables don’t always map to objects in applications very well
  - Not ideal for unstructured data: data streams or video, audio etc.

RDBMSs Lesson learned (1)

- Fantastic method for data storage
  - We will always use RDBMS!
- Scaling is an issue
  - How to accommodate more and more users
- Limits to the size of a physical server
- Joins are expensive (time to retrieve inner joins, etc.)
- Modern industry uses a combination of RDBMSs and Non-RDBMSs
  - Select the appropriate DB for each problem!

RDBMSs Lesson learned (1)

- Use cases for RDBMSs:
  - Store information about our customers?
    - ✓ Yes
  - Store data in accounting software?
    - ✓ Yes
  - Library software to organise books?
    - ✓ Yes
  - British Library software to store all of their digital treasures?
    - X No
  - Data streams of 10GBs every 10 minutes for data storage and analysis?
    - X No

What is NoSQL?

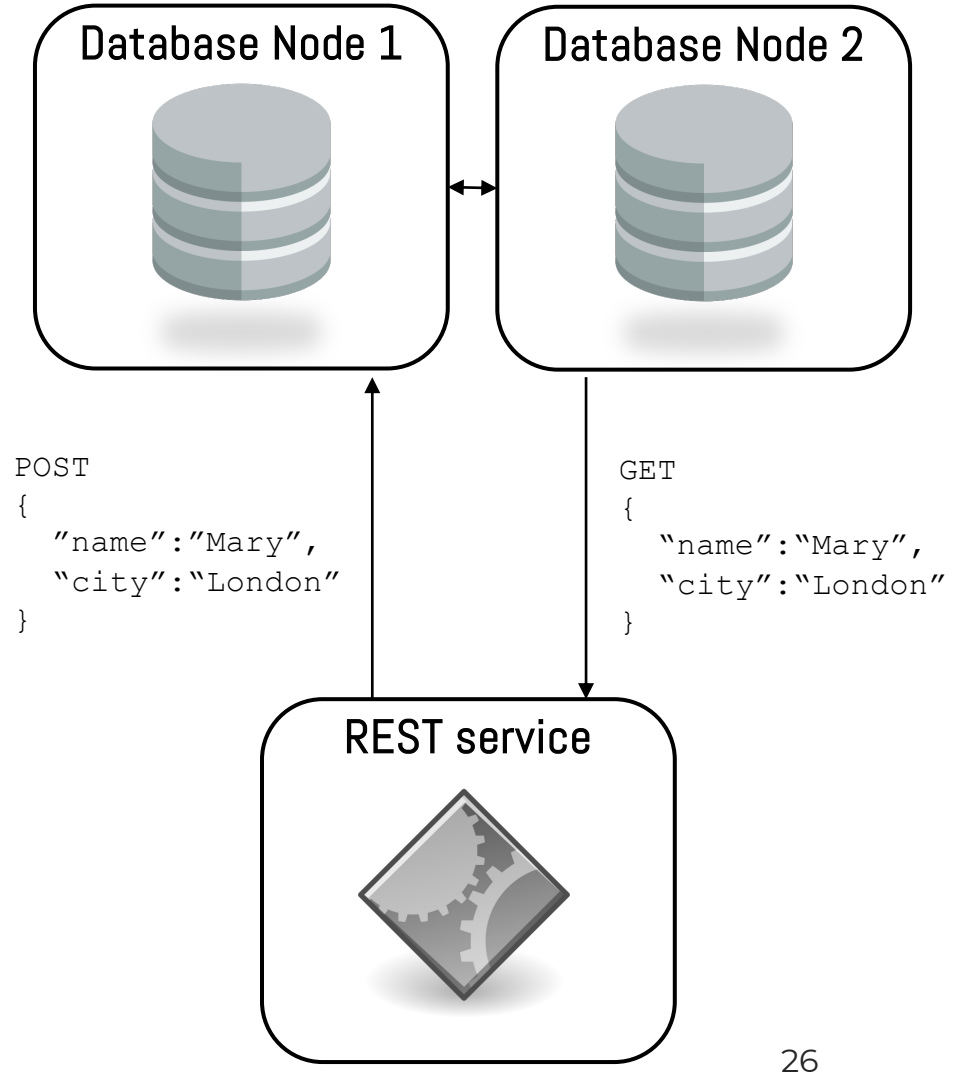
- An alternative way to store data in a database system
  - **Document databases** (e.g. MongoDB)
    - Data is stored in the form of free-form JSON structures or “documents”
  - **Key-value stores** (e.g. Redis)
    - Free-form values that are accessed in the database by way of keys and values
  - **Wide column stores** (e.g. Cassandra)
    - Data is stored in columns instead of rows as in a conventional SQL system
  - **Graph databases** (e.g. Neo4j)
    - Data is represented as a network or graph of entities and their relationships

NoSQL characteristics

- A schema-less data storage
  - Quite flexible
- Fast access to the data:
  - You care more about speed and simplicity of accessing data (reads-writes) than reliable transactions or consistency.
- Storing large volumes of data
  - Querying TB/PT of data

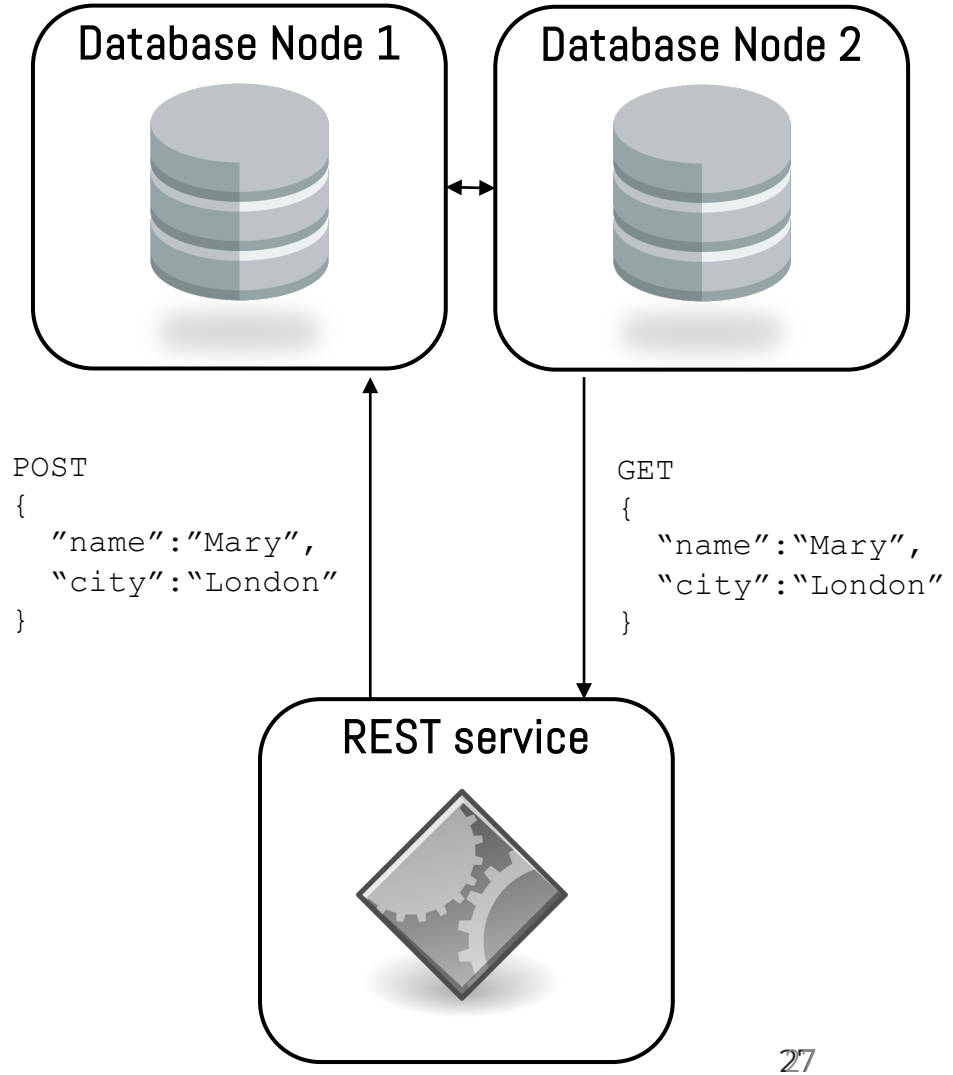
Distributed systems

- Our App becomes popular with thousands or even millions of users…
  - We need a distributed database system.
- The service will need to talk to whichever machine to get the data.



Looks good but…

- What are the problems?
  - **C**onsistency:
    - All nodes return the same results
  - **A**vailability:
    - A request returns a result
  - **P**artition tolerance:
    - Can survive partitions (failures)



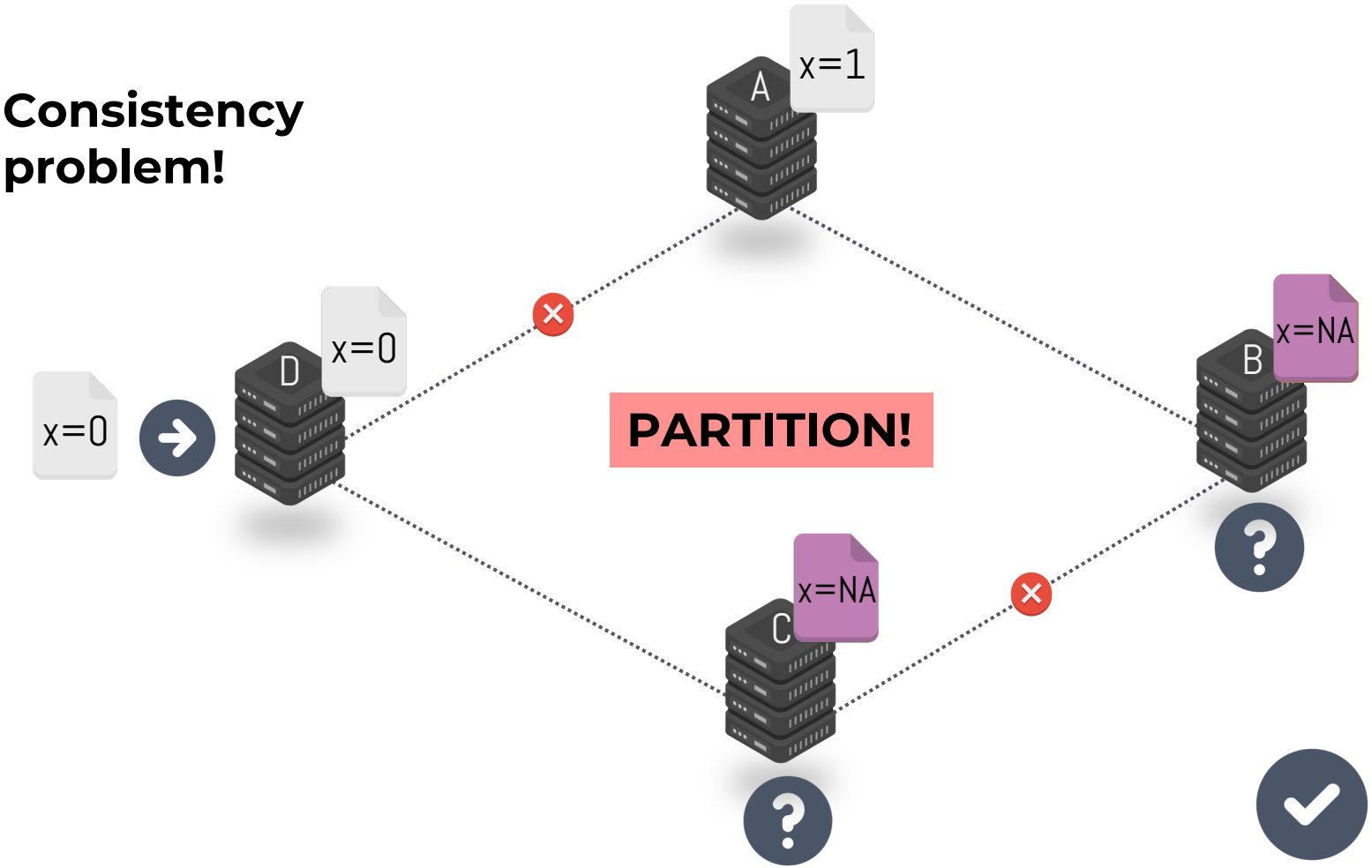
CAP Theorem

- **CAP Theorem** states that **a distributed database system can only have two of the following three options**:
    1. **Consistency**: Each node provides the most recent state or does not provide a state at all.

       * No two nodes will return a different state at a given time
    2. **Availability:** Every node has constant **read** and **write** access
    3. **Partition Tolerance:** Can survive a partitions

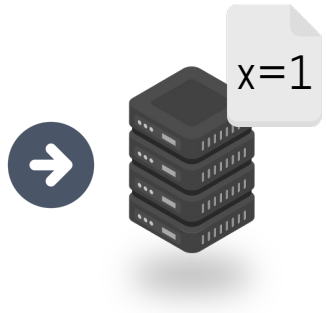
       * Partition: Failures can happen (communication)

We can only have two out of three properties...

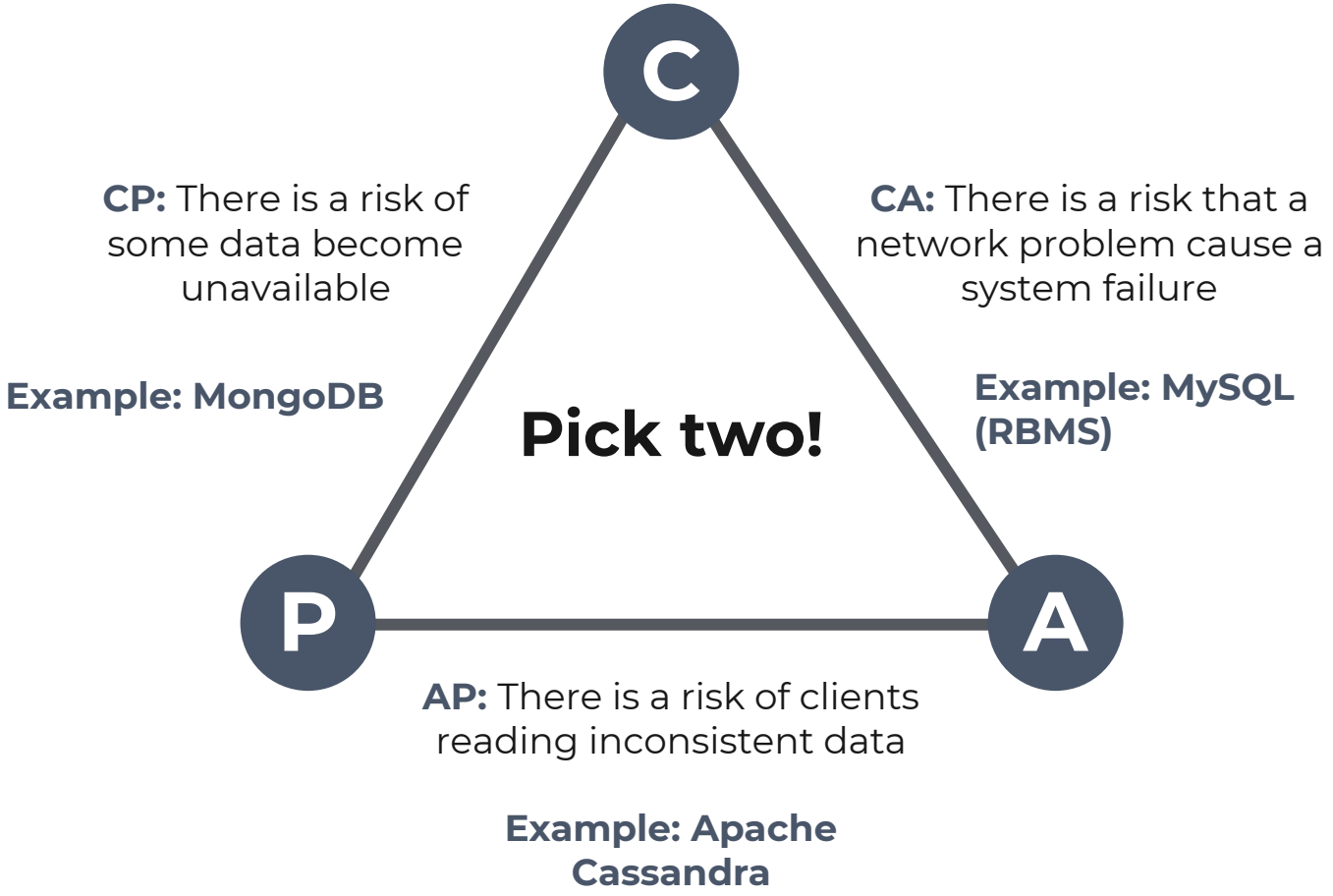


We can only have two out of three properties...

What if I want to be consistent and available?



- You cannot survive partitions!
- **You cannot be distributed!**



What is **BASE**?

- A hack of CAP!
- NoSQL support BASE:
  - **BAS**ically **A**vailable
    - DB is always available as per the CAP theorem
    - DB **is inconsistent but becomes consistent over time!**
    - DB supports partition tolerance (failures)
  - **S**oft state
    - Even without an input, the system state may change
  - **E**ventual we will have consistency
    - This m eans that the system will become consistent over time

What is an example of a BASE application?

- Instagram example: No one would mind if posts were inconsistent within their social network for a short period – but always available.
- It is more important to get an immediate response than wait to have a consistent state of users' information.

Quiz!

- Assume you have 20TB of data that is sorted by city and data can be distributed!
- What is the most efficient way to reorganize the data if you are always searching by `CITY`?

```
SELECT * FROM CUSTOMERS WHERE CITY=<a_city>;
```

CUSTOMERS

ID	NAME	AGE	GENDER	CITY
...	...	...	...	...
101	John	25	M	London
102	Nick	30	M	London
103	Peter	45	M	London
104	Mary	16	F	Toronto
105	Anna	30	F	Toronto
106	Kate	39	F	Toronto
...	...	...	...	...

Select using a partition key!

```
SELECT * FROM CUSTOMERS WHERE CITY='Toronto';
```

CUSTOMERS

ID	NAME	AGE	GENDER	CITY
...	...	...	...	...
101	John	25	M	London
102	Nick	30	M	London
103	Peter	45	M	London
104	Mary	16	F	Toronto
105	Anna	30	F	Toronto
106	Kate	39	F	Toronto
...	...	...	...	...

- What if you already know that you need to search in the range of 104 to 106?
- And you somehow know that 104 to 106 are in server 5?

To consider: How do tables look like in NoSQL?

RDBMS

CUSTOMERS				
ID	NAME	AGE	GENDER	CITY
1	John	25	M	London
2	Nick	30	M	London
3	Peter	45	M	London
4	Carmen	16	F	Toronto
5	Anna	30	F	Toronto
6	Tibi	39	F	Toronto

CUSTOMERS				
1	{name:John}	{age:25}	{gender:M}	{city:London}
2	{name:Nick}	{age:30}	{gender:M}	{city:London}
3	{name:Peter}	{age:45}	{gender:M}	{city:London}
4	{name:Carmen}	{age:16}	{gender:F}	{city:Toronto}
5	{name:Anna}	{age:30}	{gender:F}	{city:Toronto}
6	{name:Tibi}	{age:39}	{gender:F}	{city:Toronto}

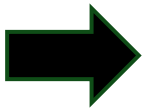
CUSTOMERS				
1	{name:John}	{age:25}	{gender:M}	{city:London} *3
2	{name:Nick}	{age:30}	{gender:M}	
3	{name:Peter}	{age:45}	{gender:M}	
4	{name:Carmen}	{age:16}	{gender:F}	{city:Toronto} *3
5	{name:Anna}	{age:30}	{gender:F}	
6	{name:Tibi}	{age:39}	{gender:F}	

Query first design! Example of querying by GENDER

Query first design

Names per GENDER				
M	{name:John}	{age:25}	{id:1}	{city:1}
M	{name:Nick}	{age:30}	{id:2}	{city:1}
M	{name:Peter}	{age:45}	{id:3}	{city:1}
F	{name:Carmen}	{age:16}	{id:4}	{city:2}
F	{name:Anna}	{age:30}	{id:5}	{city:2}
F	{name:Tibi}	{age:39}	{id:6}	{city:3}

Cities per ID			
1	{name:London}	{population:9}	{country:UK}
2	{name:Toronto}	{population:3.2}	{country:Canada}
3	{name:Athens}	{population:3}	{country:Greece}



Cassandra Data

	Characters per GENDER			
M*3	{name:John}	{age:25}	{id:1}	{city:1}
	{name:Nick}	{age:30}	{id:2}	{city:1}
	{name:Peter}	{age:45}	{id:3}	{city:1}
F*3	{name:Carmen}	{age:16}	{id:4}	{city:2}
	{name:Anna}	{age:30}	{id:5}	{city:2}
	{name:Tibi}	{age:39}	{id:6}	{city:3}

	Cities per ID		
1	{name:London}	{population:9}	{country:UK}
2	{name:Toronto}	{population:3.2}	{country:Canada}
3	{name:Athens}	{population:3}	{country:Greece}

Apache Cassandra

Why to use Cassandra?

- Easy to learn!
- It is ideal when we don't care about consistency → BASE
- Lightning-fast writes and fast reads
- Extremely well when distributed!
  - Some of the largest production deployments include:
    - Apple: Over 75,000 nodes storing over 10 PB of data
    - Netflix: 2,500 nodes, 420 TB, over 1 trillion requests per day)
    - eBay (over 100 nodes, 250 TB).

Apache Cassandra

- How does it work?
  - Each query should be handled by one table in the database
  - **Increased performance:** We query for data in the same table(s)!
- Partitioning:
  - Every data point is assigned a partition key
  - The same partition key will be stored in the same node in a cluster
  - For example, the following data will always be stored in the same node:
    - **Get all female employees of our company**

PERSONS per GENDER				
F*3	{name:Carmen}	{age:16}	{id:4}	{city:2}
	{name:Anna}	{age:30}	{id:5}	{city:2}
	{name:Tibi}	{age:39}	{id:6}	{city:3}

Partition keys

- A strategy that helps us organise data in nodes
  - **If the partition key is unique,** then the data will be distributed in a cluster.
    - (does not make any difference)
  - **If the partition key is not unique**, then data with the same key will always be in the same node, making querying a fast and efficient process.
    - Cassandra will look only in the particular node for this data, skipping the other nodes.
    - In our example, Gender is the partition key and ID is the primary key (as in SQL)

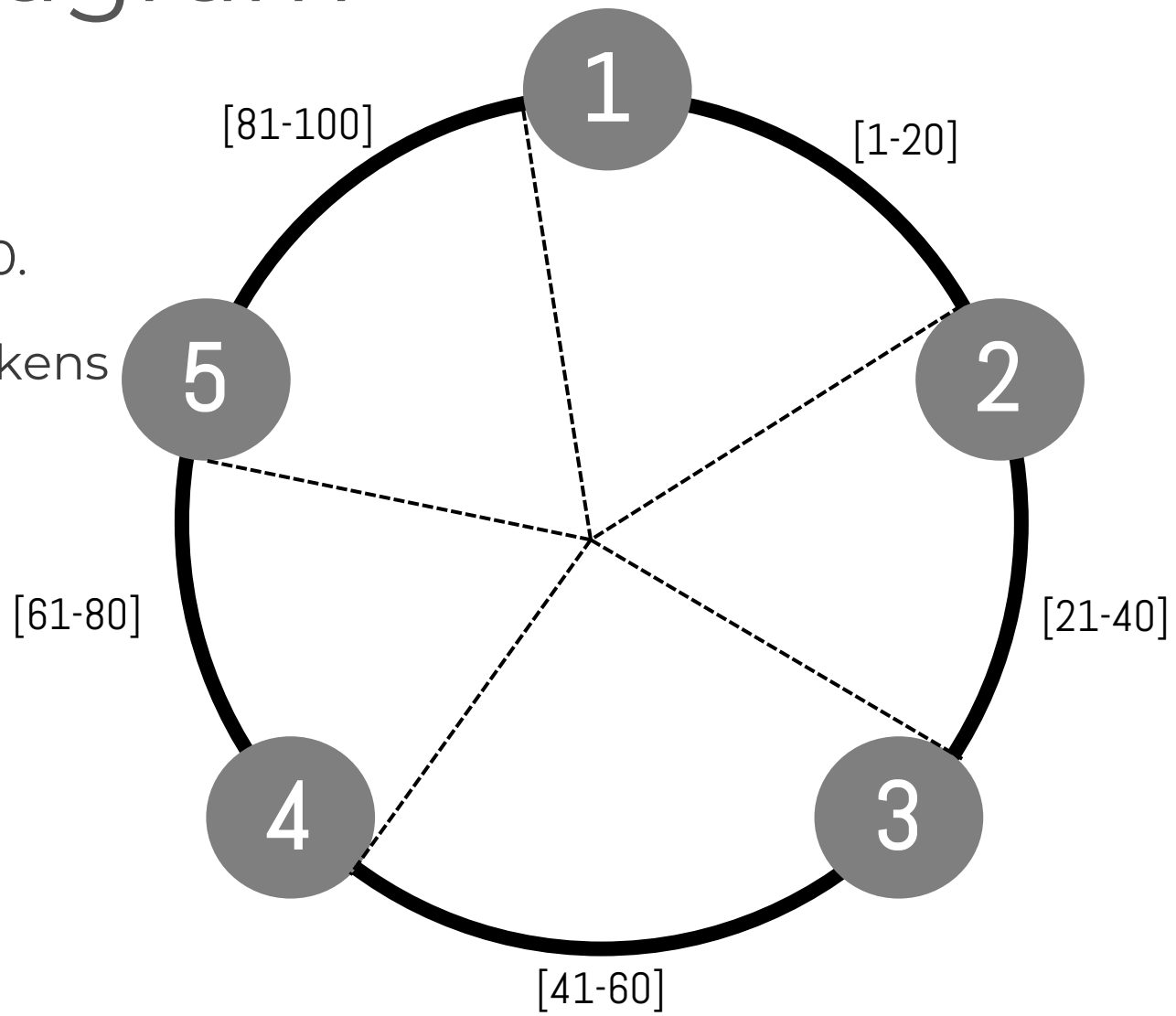
PERSONS per GENDER				
M*3	{id:1}	{name:John}	{age:25}	{city:1}
	{id:2}	{name:Nick}	{age:30}	{city:1}
	{id:3}	{name:Peter}	{age:45}	{city:1}
F*3	{id:4}	{name:Carmen}	{age:16}	{city:2}
	{id:5}	{name:Anna}	{age:30}	{city:2}
	{id:6}	{name:Tibi}	{age:39}	{city:3}

How Cassandra inserts data?

- Cassandra decides in which node to insert data using a **hash function**
  - Remember, if we use a partition key, data with the same partition key should always be in the same node!
- A hash function is a way to get a unique value (key) out of a particular input
  - For example, myHash is a function that generates a key:
    - myHash(”stelios”) = 5
    - myHash(“mary”) = 7
    - myHash(”stelios”) = 5
      - The hash of the same input is always the same output!
  - In Cassandra:
    - Input is the partition **Key / Output** is **a node** to insert data
      - A node that accepts a range of keys

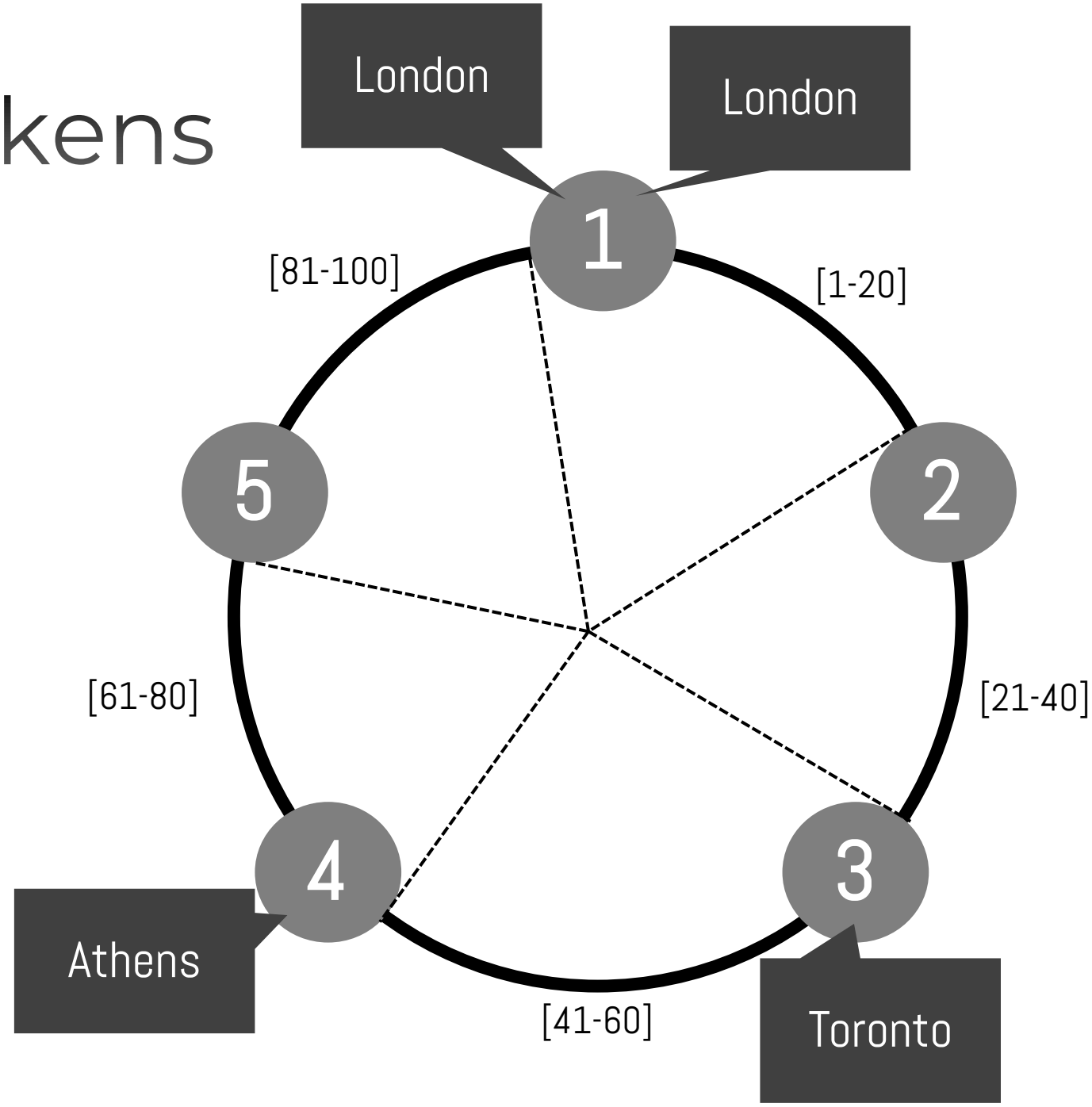
Cassandra Ring diagram

- Let us assume that our hash function generates numbers between 1 and 100.
- Each node takes a specific range of tokens
- For example:
  - Node 1: Will take tokens [1-20]
  - Node 2: Will take tokens [21-40]
  - Node 3: Will take tokens [41-60]
  - Node 4: Will take tokens [61-80]
  - Node 5: Will take tokens [81-100]



Example with tokens

- Partition key: London
  - HashFunction() = 10
- Partition key: Athens
  - HashFunction() = 80
- Partition key: Toronto
  - HashFucntion() = 50
- Partition key: London
  - HashFunction() = 10



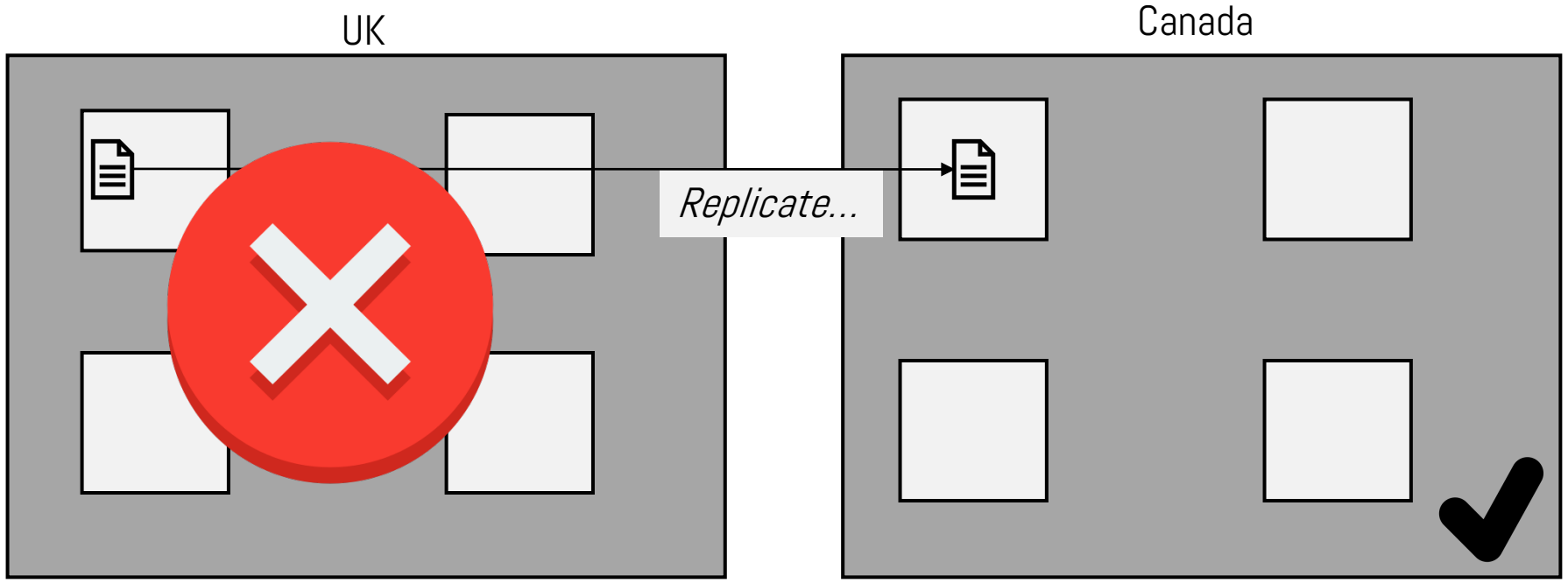
Replication in Distributed DBs

- In different servers on the same rack
- In different servers on a different rack
- In different servers on a different Datacenter
- In different servers on a datacentre in different regions

Server

Replication in Cassandra

- Cassandra can be used to support multiple datacenters
  - Allows to store data in different regions around the world and store backups in different regions
  - Easy to store and replicate, e.g. in different racks in different regions



How to interact with Cassandra?

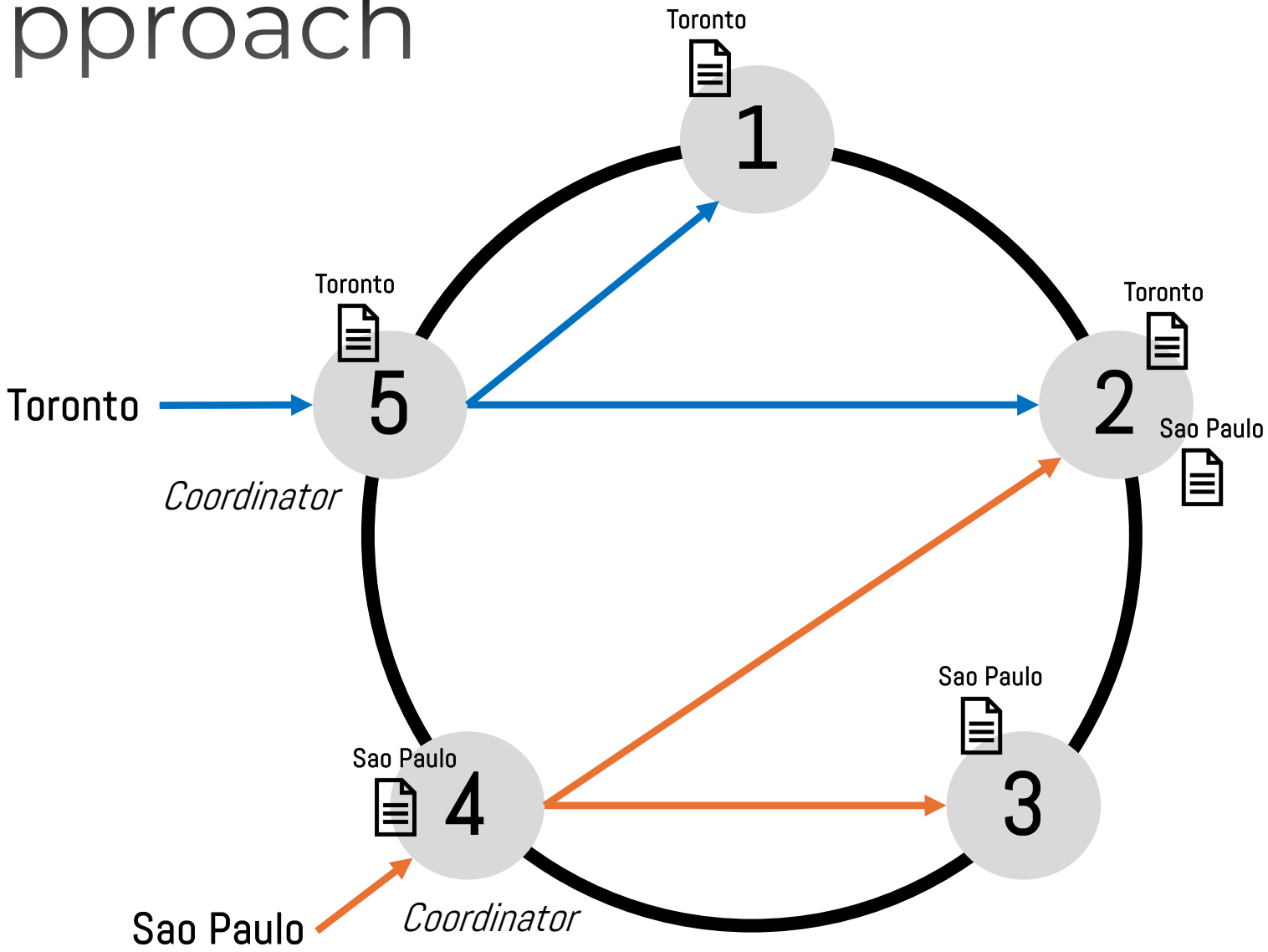
- Use the command line interface:
  - Called CQL
- Use Cassandra driver to connect from Python:
  - Connect and execute commands automatically

```
# Create a new KEYSPACE (Database)
CREATE KEYSPACE stelios_key_space WITH REPLICATION=
        {'class':'SimpleStrategy', replication_factor='3'};

# Create a new table with a partition key CITY_NAME and PRIMARY KEY
CITY_NAME,ID
CREATE TABLE CHARACTERS_BY_CITY
   (CITY_NAME TEXT, ID INT, NAME TEXT, PRIMARY KEY(CITY_NAME,ID));
```

Peer to Peer approach

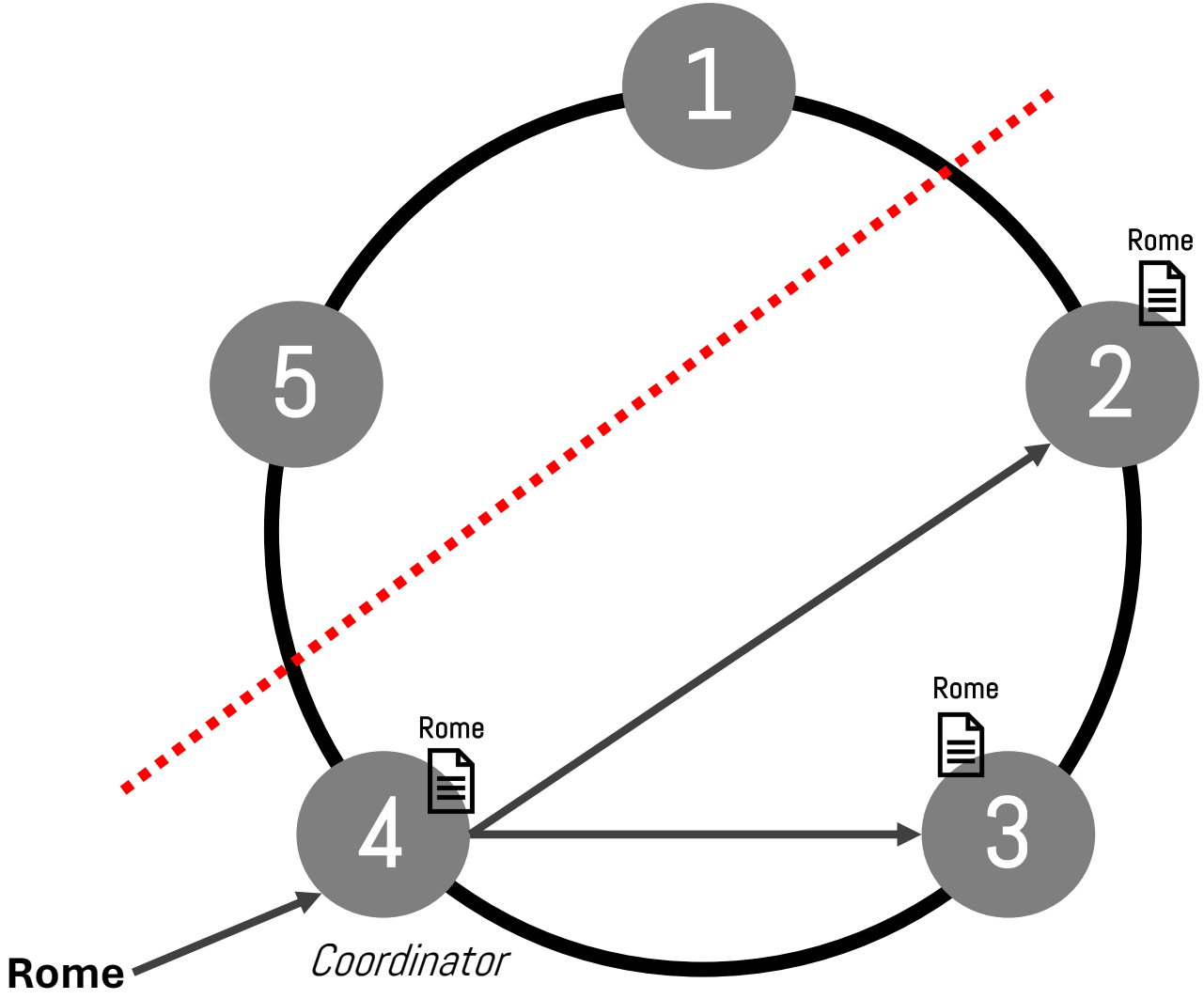
- How to write data?
  - Replication factor: 3



Partition tolerance

- How to write data?
  - Replication factor: 3

This would be successful!

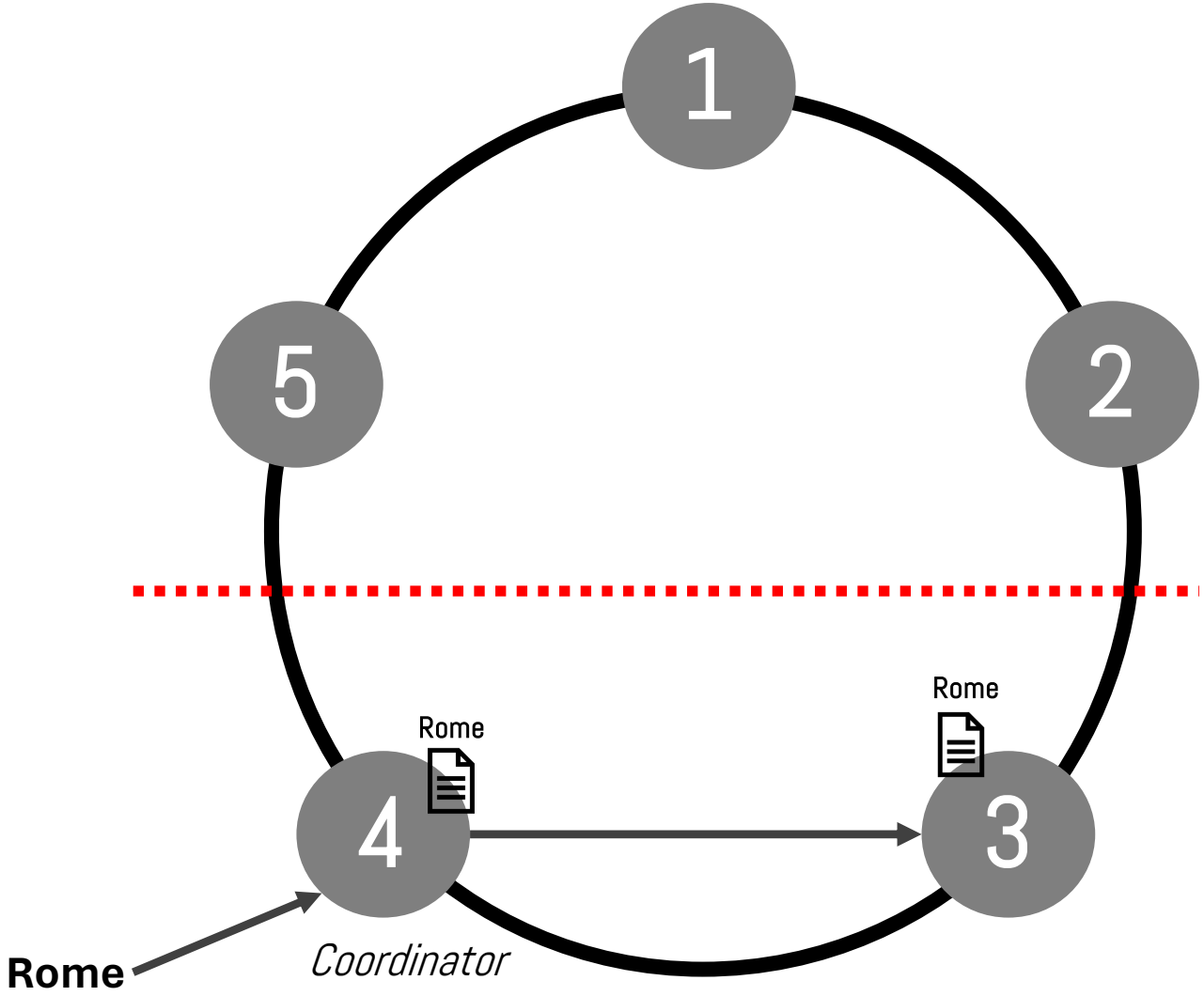


Partition tolerance

- How to write data?
  - Replication factor: 3

This would be unsuccessful!

We need to store two copies!

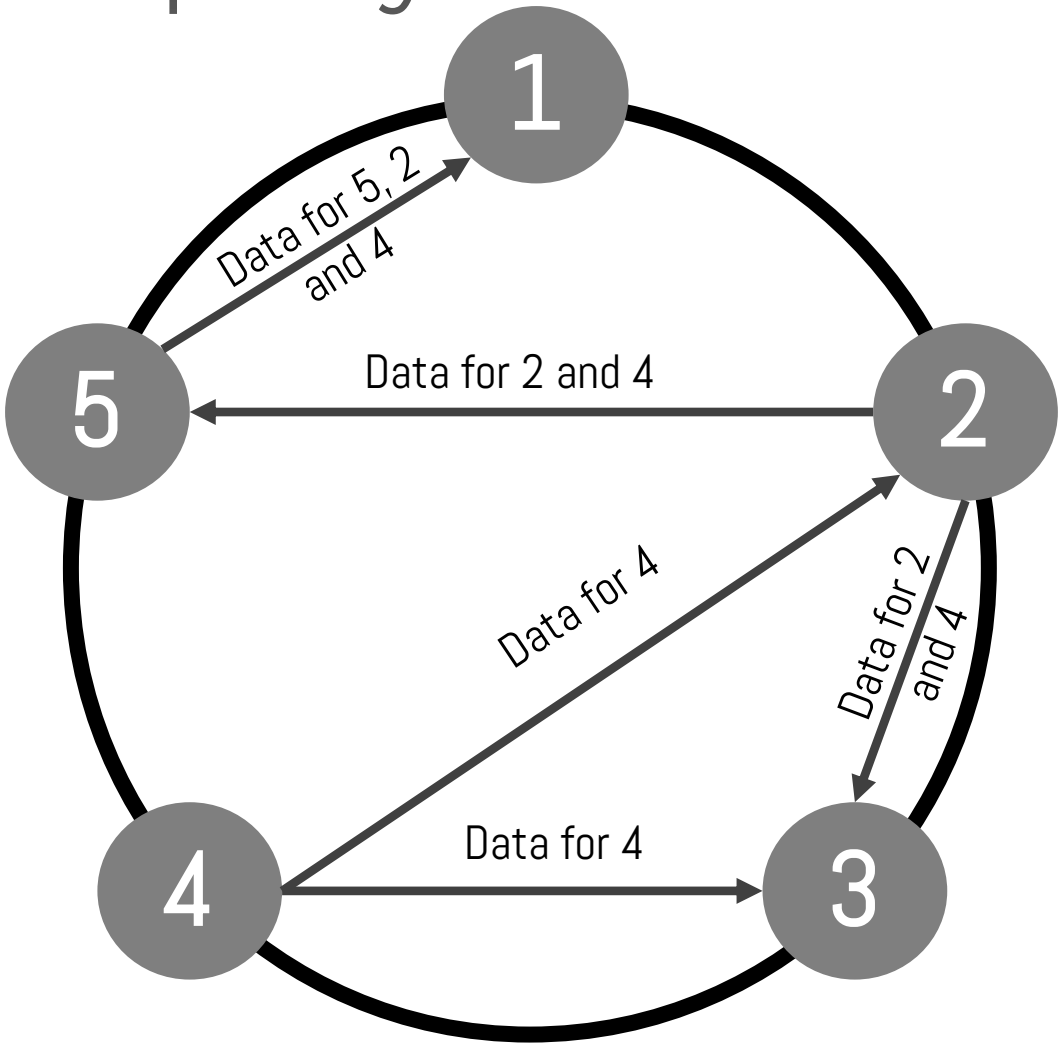


How Cassandra nodes know for each other in a decentralized topology

- **Cassandra Snitch:** a service to understand the topology of the cluster
  - Where is each node?
    - Datacenter locations and racks
  - What is the most responsive node of our cluster?
    - Faster to read and write data in terms of locality, e.g., a client in the UK will write and read data faster in nodes in a UK datacenter.
- **Simple Snitch:**
  - It’s suitable for testing: All nodes are in the same datacenter and the same rack
- **Property file snitch:**
  - It’s suitable for production and contains information about nodes[IP – DC – Rack]
    - Apple has 25.000 nodes running in a cluster

Cassandra: Gossiping Property file

- For each node, we specify the Cassandra property file
  - We use a file for each datacenter: e.g. DC number and rack number
  - We use a Gossiping protocol to distribute this information.
  - Nodes know about other nodes in the cluster:
    - Status, health current state, addresses, schema version, tokens etc.



What about searching in Cassandra?

- Apple: Over 75,000 nodes storing over 10 PB of data…
  - That’s big data! NoSQL is ideal ☺
- How do we search?
- One by one?
  - Usually, we search a small volume of data in a large pool stored somewhere in a huge cluster!

How does Cassandra check if an element is present?

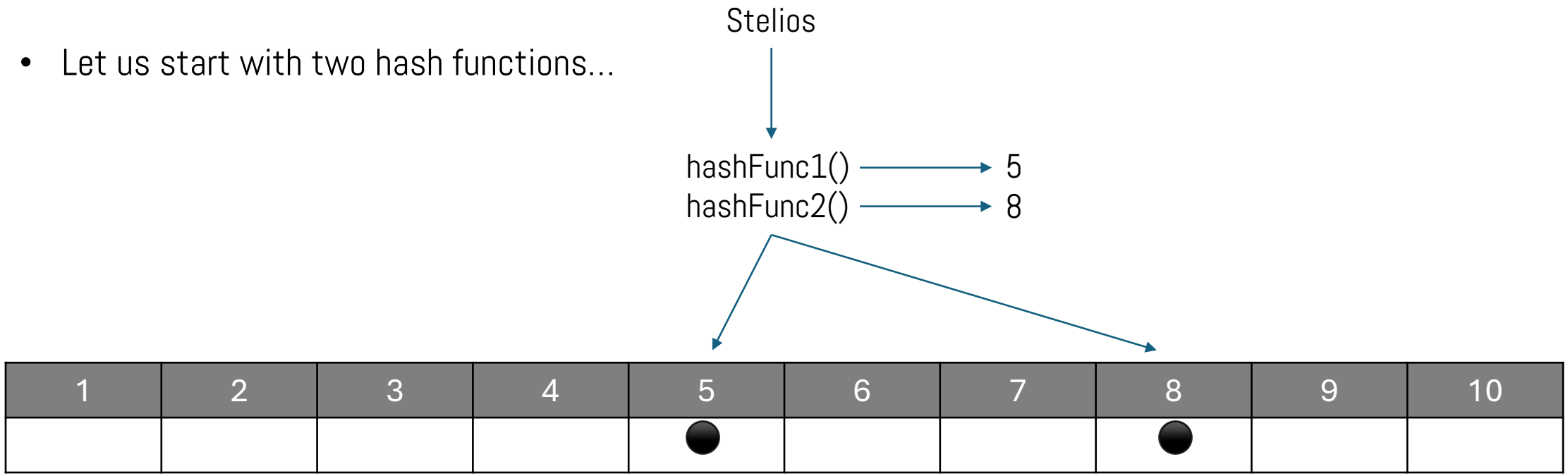
- It uses a very famous algorithm called **Bloom filter**:
  - An algorithm is designed to be extremely fast at the cost of risking returning false positives.
- Cassandra uses bloomfilters to test if any of its tables is **likely** to contain the requested partition key or not without reading all data
  - Avoid expensive I/O operations
- Is London in my dataset? (London is our partition key)
  - **FALSE** → 100% the value does not exist!
  - **TRUE** → Maybe is there...

Bloom filter

- A Bloom filter is a data structure designed to tell you, fast and memory-efficiently, whether an element is not present in a set.
- The price paid for this efficiency is that a Bloom filter is a **probabilistic data structure**:
  - An element either is not in the dataset or it may be in the set.

Example: Insert data

- Let us start with two hash functions…



What is the ●?

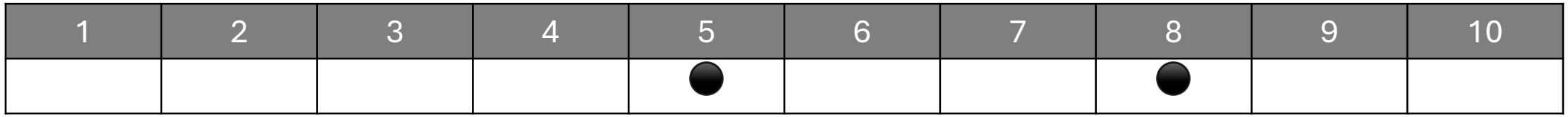
- It could be 1 or 0
- It's a Boolean operator

Example: Query data

Is "Stelios" in the dataset?

Maybe!

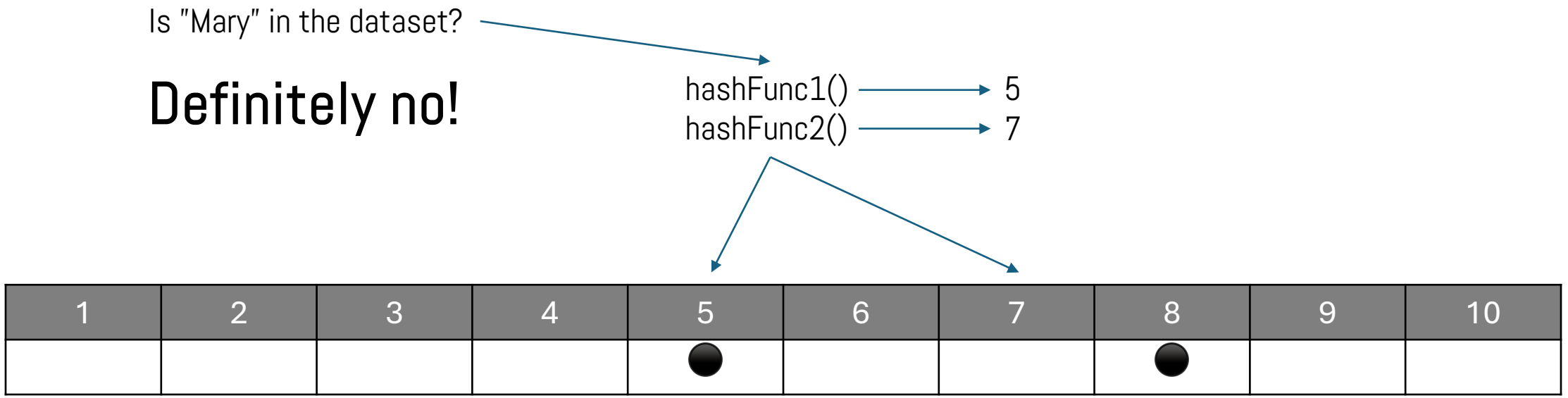
hashFunc1() → 5
hashFunc2() → 8



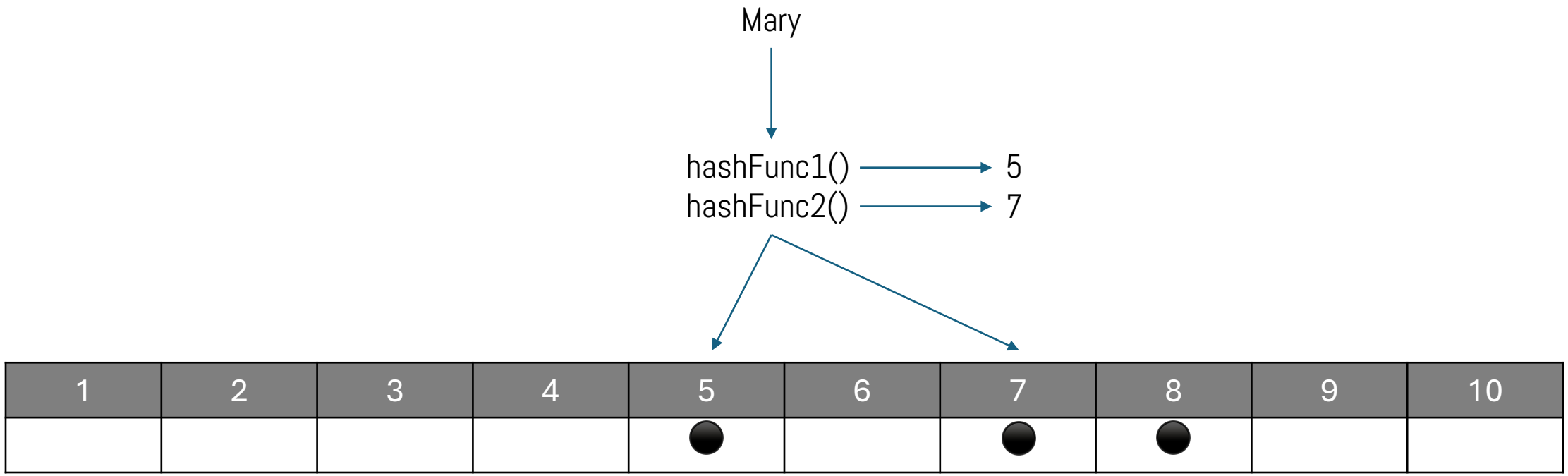
Example: Query data

Is "Mary" in the dataset?

Definitely no!



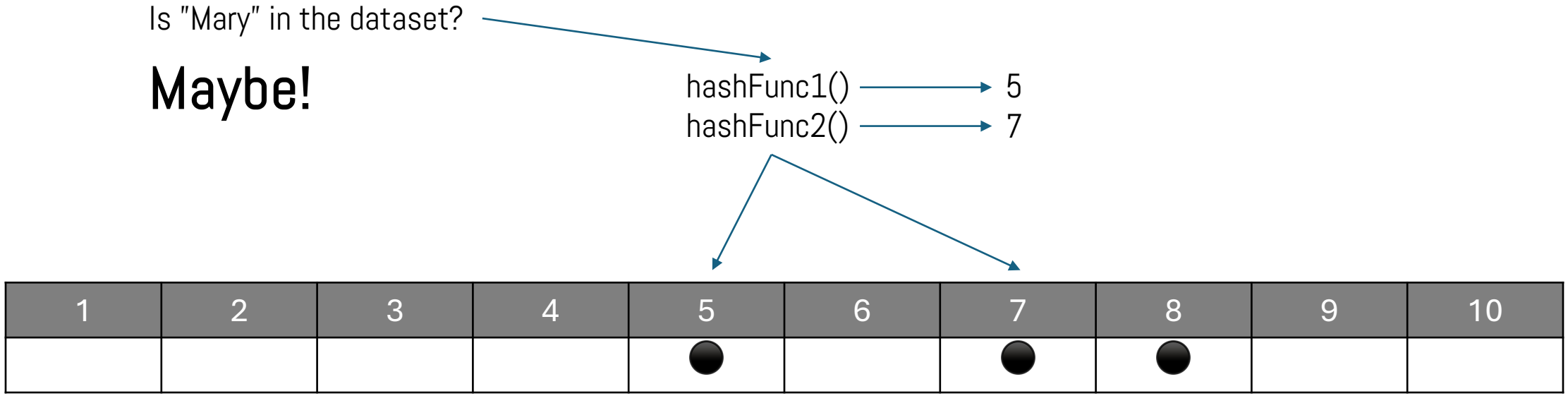
Example: Insert data



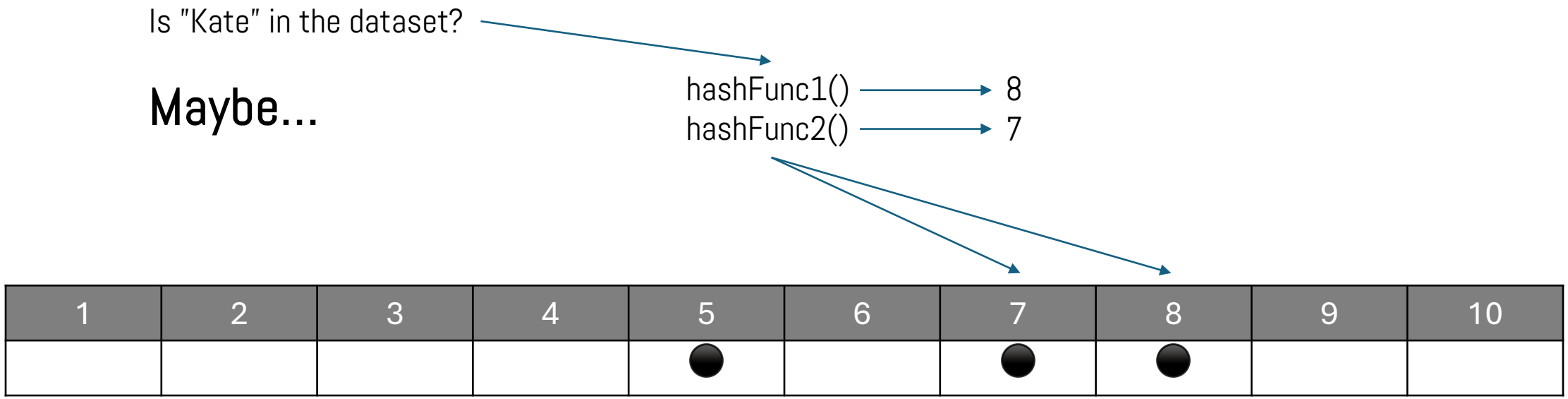
Example: Query data

Is "Mary" in the dataset?

Maybe!



Example: Query data



Now Cassandra will employ sophisticated techniques to look for “Kate”

In our case “Kate” is not a partition key...

Want to learn Cassandra?

- If you like to learn more about Cassandra
  - Cassandra: The Definitive Guide, (Revised) Third Edition, 3rd Edition, January 2022

Let’s summarise

When to use NoSQL

- You need to work on massive amount of data.
- You require fast writes.
- There is no need for joins or aggregates.
- There is a requirement to integrate with Big Data, Hadoop, Hive, and Spark.
- There is a need for a distributed application.

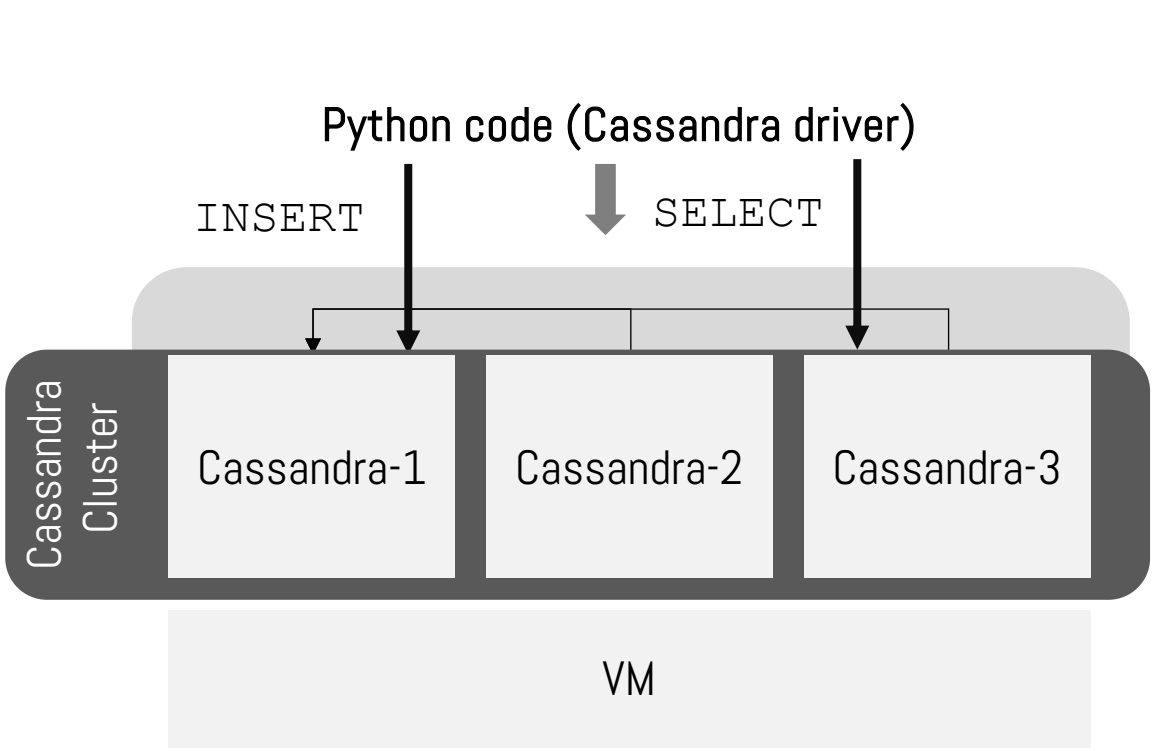
When to use RDBMS

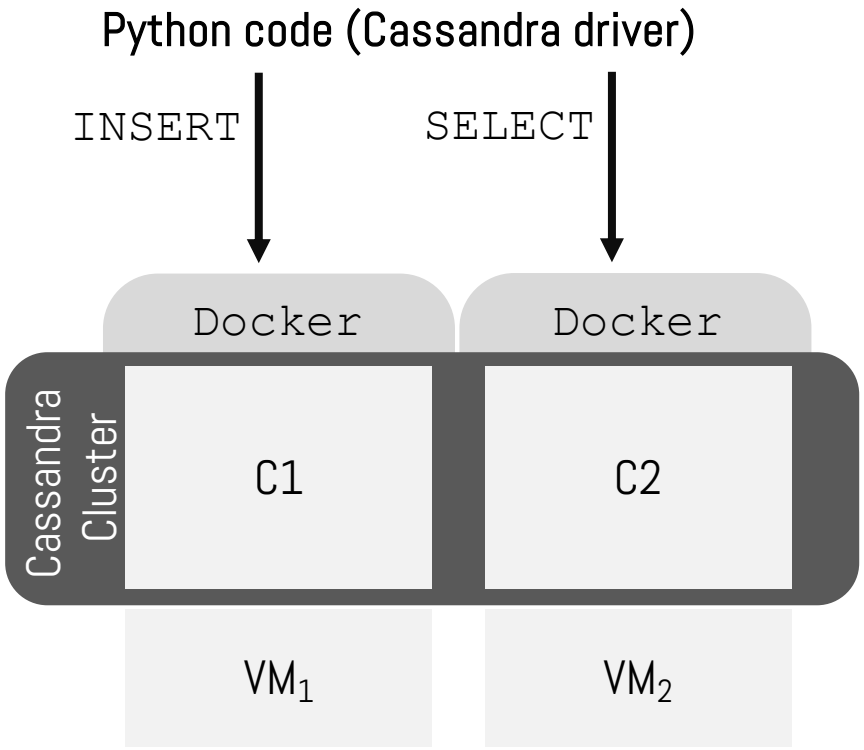
- You need to store volumes of data across racks of clusters.
- Data needs to be in a relational format.
- You have a strong requirement for ACID properties (use RDBMS).
- You want to use aggregate functions.

Lab 7

Big Data Analytics

Apache Cassandra Cluster





To do

- Install Cassandra clusters (in same and different GCP VMs)
  - 3 Cassandra nodes
  - Follow the tutorial
- Run simple scripts to interact with Apache Cassandra
  - CREATE A KEYSPACE
  - INSERT DATA
    - SQL LIKE DATA…
    - JSON LIKE DATA…
  - SELECT DATA and create an index to access key-value data
- Develop a Python application to interact with the Cassandra Cluster

Lab activities

- Complete lab 7 activities and exercises.
- Use the GCP to run exercises in Cassandra.