# Emerging Paradigms for Big Data

Cassandra

Cassandra + Kafka

Fátima Leal

DCT DEPARTAMENTO CIÊNCIA E TECNOLOGIA

UPT UNIVERSIDADE PORTUCALENSE

# Content

- Introduction to Cassandra
- Distributed vs centralised
- Elastic Scalability
- High availability and fault tolerance
- Tuneable Consistency
- Brewer’s CAP Theorem
- Examples – Exercises
- Cassandra Connector

# Introduction to Cassandra

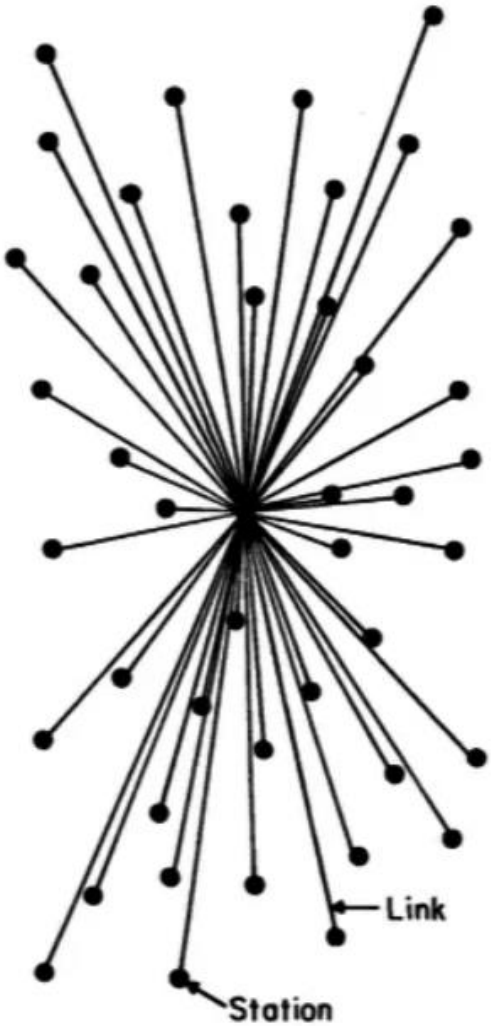
# Introduction

- **Apache Cassandra** is an open source, distributed, decentralized, elastically scalable, highly available, fault-tolerant, tuneably consistent, row-oriented database.

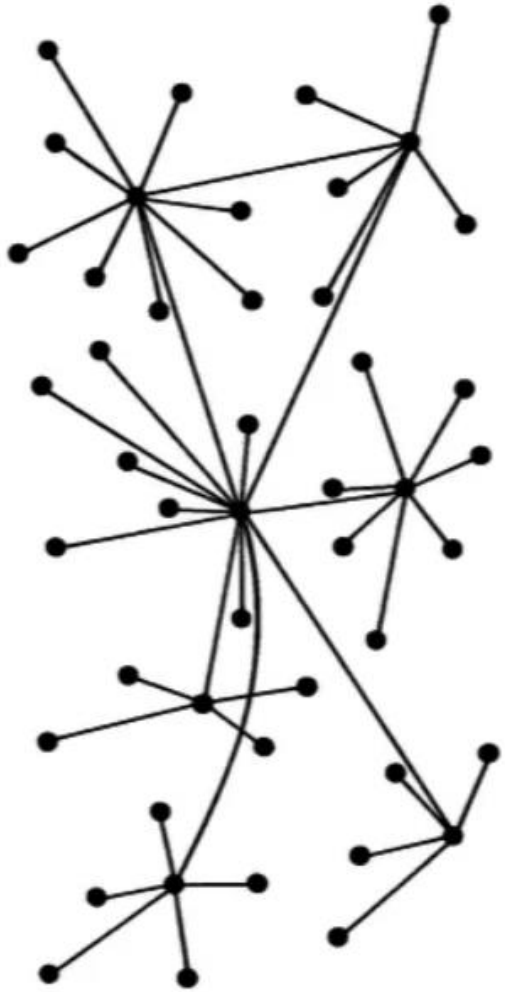
- Cassandra uses a query language similar to SQL.

- **Created at Facebook**, it now powers **cloud-scale applications** across many industries.

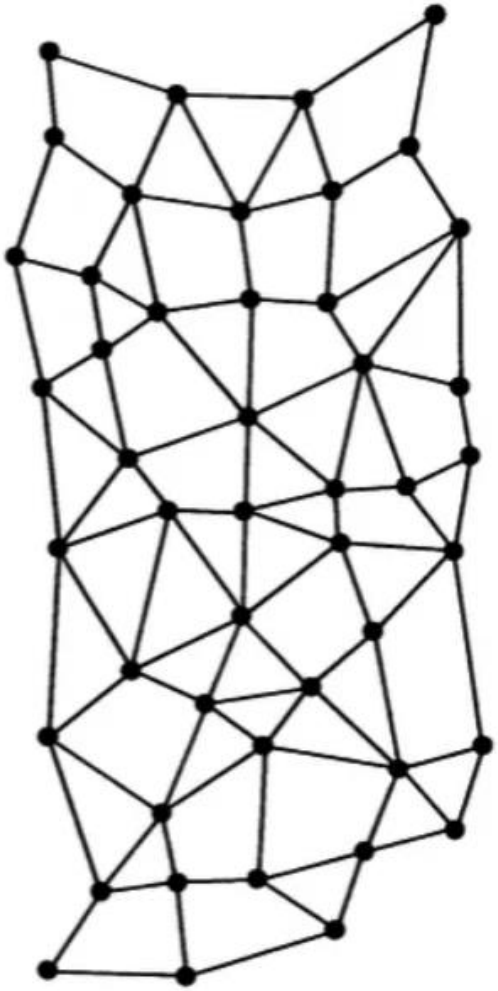
# Distributed vs Decentralised




CENTRALIZED



DECENTRALIZED



DISTRIBUTED

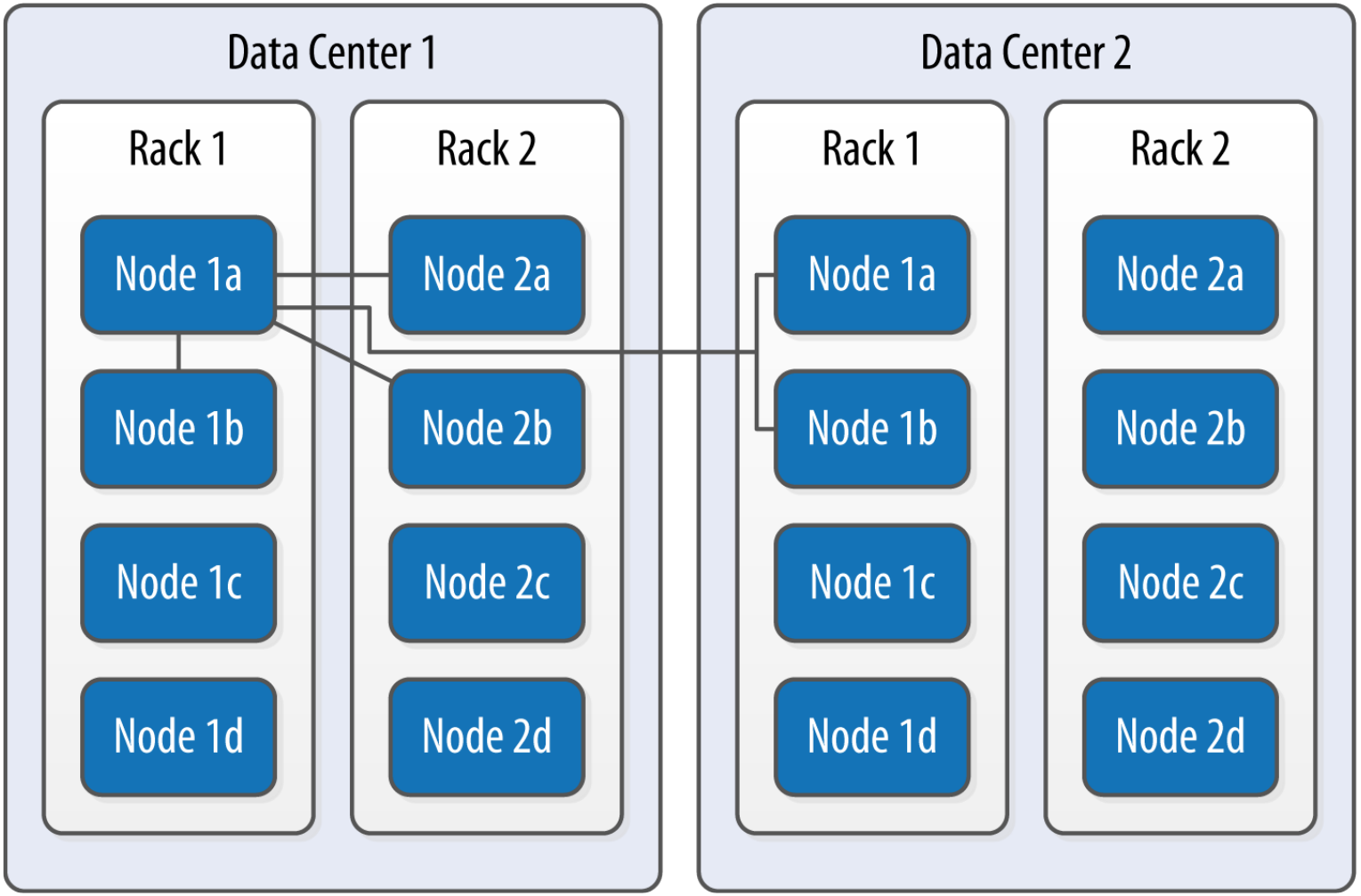


A

# Distributed vs Decentralised

- Cassandra is distributed, *i.e.*, can run on **multiple machines**
- Much of its design and code base is specifically engineered toward not only making it work across **many different machines**, but also for optimizing **performance across multiple data center racks**, and even for a single Cassandra cluster **running across geographically dispersed data centers**.
- You can confidently write data to anywhere in the cluster and Cassandra will get it.
- A **rack** is a logical **set of nodes in close proximity to each other**, perhaps on **physical machines** in a single rack of equipment.
- A **data center is a logical set of racks**, perhaps located in the same building and connected by reliable network.

# Distributed vs Decentralised



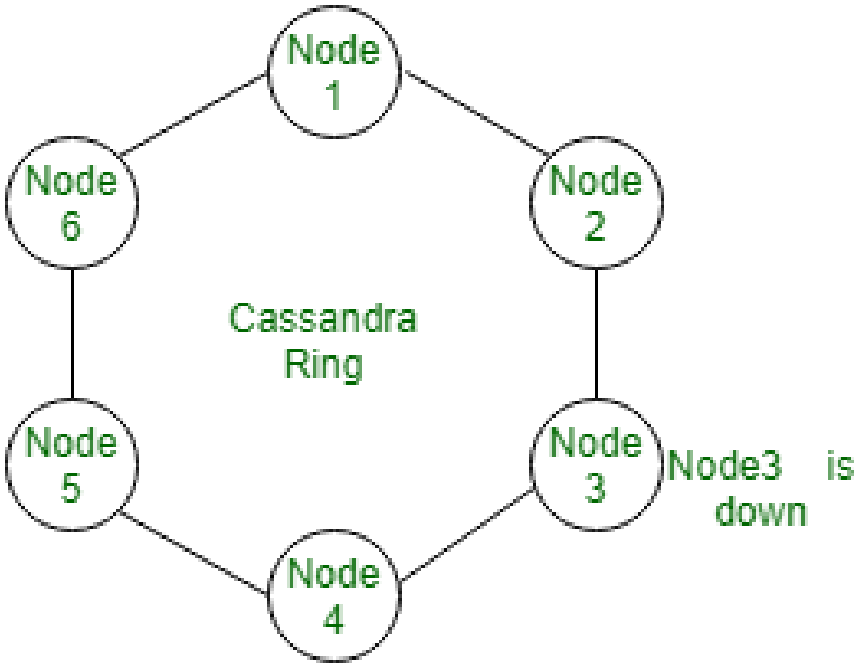
# Distributed vs Decentralised

- Once you start to scale **many other data stores** some nodes need to be set up as **primary replicas** in order to organize other nodes, which are set up as **secondary replicas.**

- Cassandra, however, is **decentralised**, meaning that every **node is identical**; no Cassandra node performs certain organizing operations distinct from any other node.

- Cassandra **features a peer-to-peer architecture** and uses a **gossip protocol** to maintain and **keep in sync a list of nodes that are alive or dead**.

# Distributed vs Decentralised




Concept of Gossip Protocol

- Gossip is a **peer-to-peer communication** protocol in which nodes periodically **exchange state information** about themselves and about other nodes they know about
- Gossip protocol runs every second and **exchange state messages** with up to three other nodes in the cluster
- Nodes exchange information about themselves and about the other nodes that they have gossiped about, so all nodes quickly learn about other nodes in the cluster

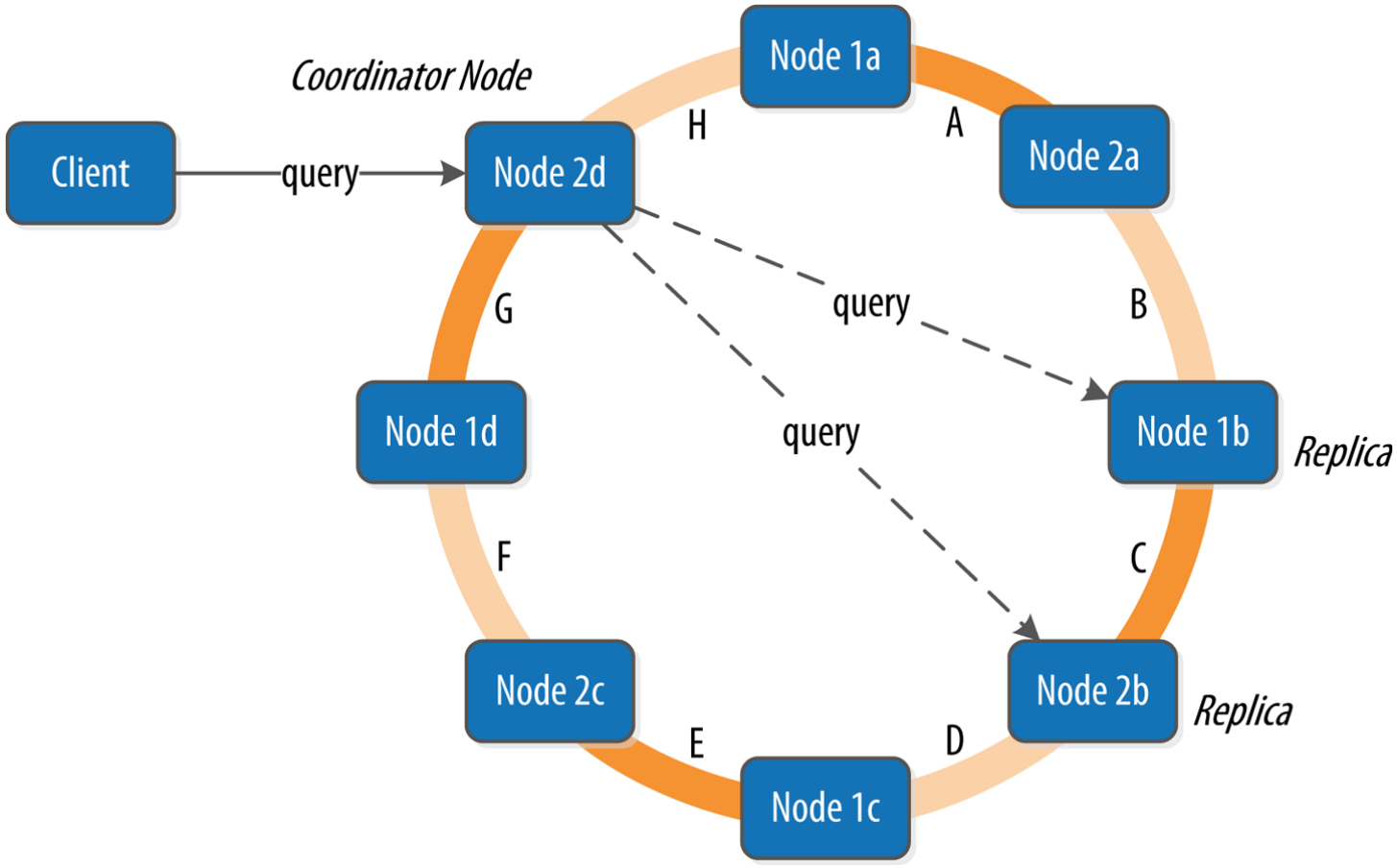
# Distributed vs Decentralised

- The fact that Cassandra is decentralized means that there is **no single point of failure.**

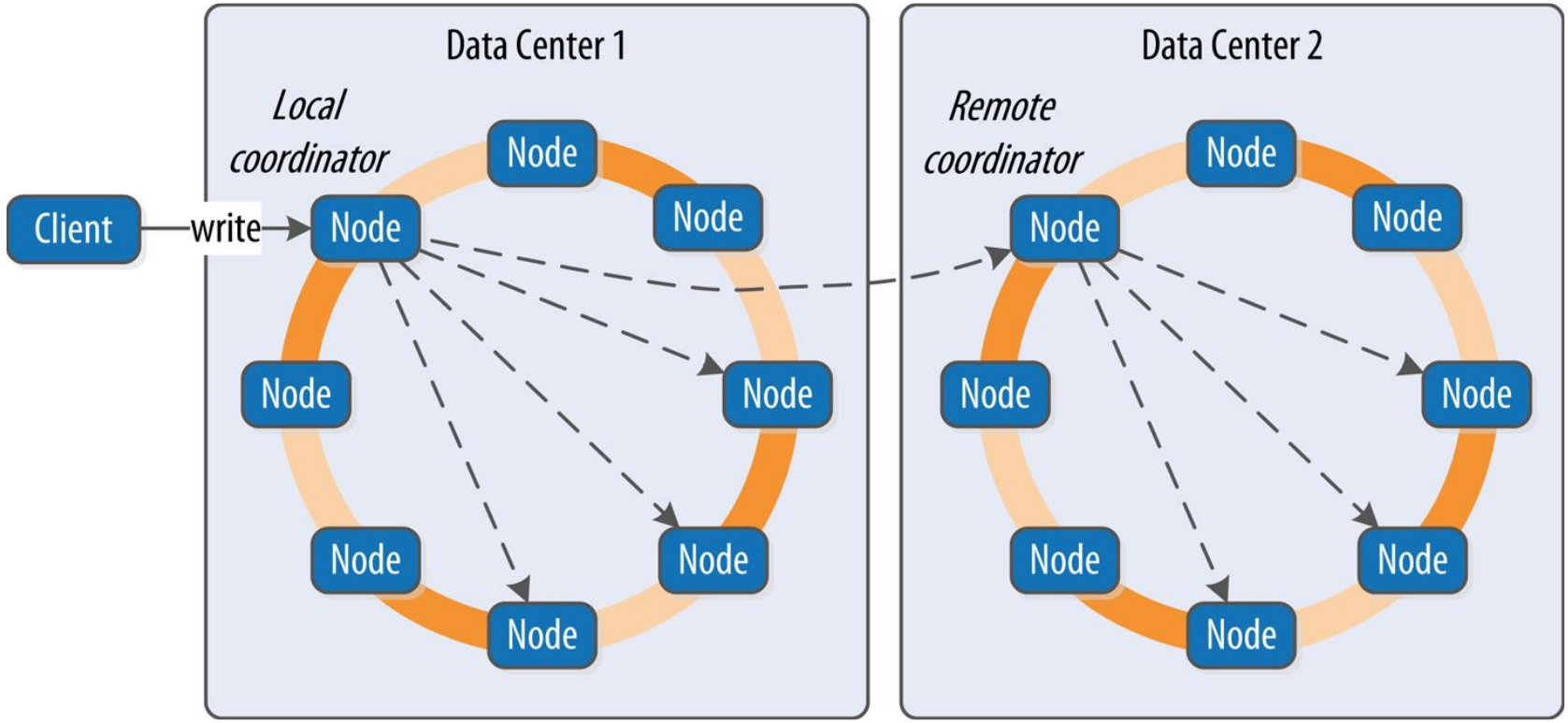
- In many distributed data solutions (such as RDBMS clusters), you **set up multiple copies of data** on different servers in a process **called replication**, which copies the data to multiple machines so that they can all serve simultaneous requests and improve performance.

- In Cassandra all the servers in this kind of cluster don’t function in the same way. You configure your cluster by designating one server as the primary and others as secondary replicas.

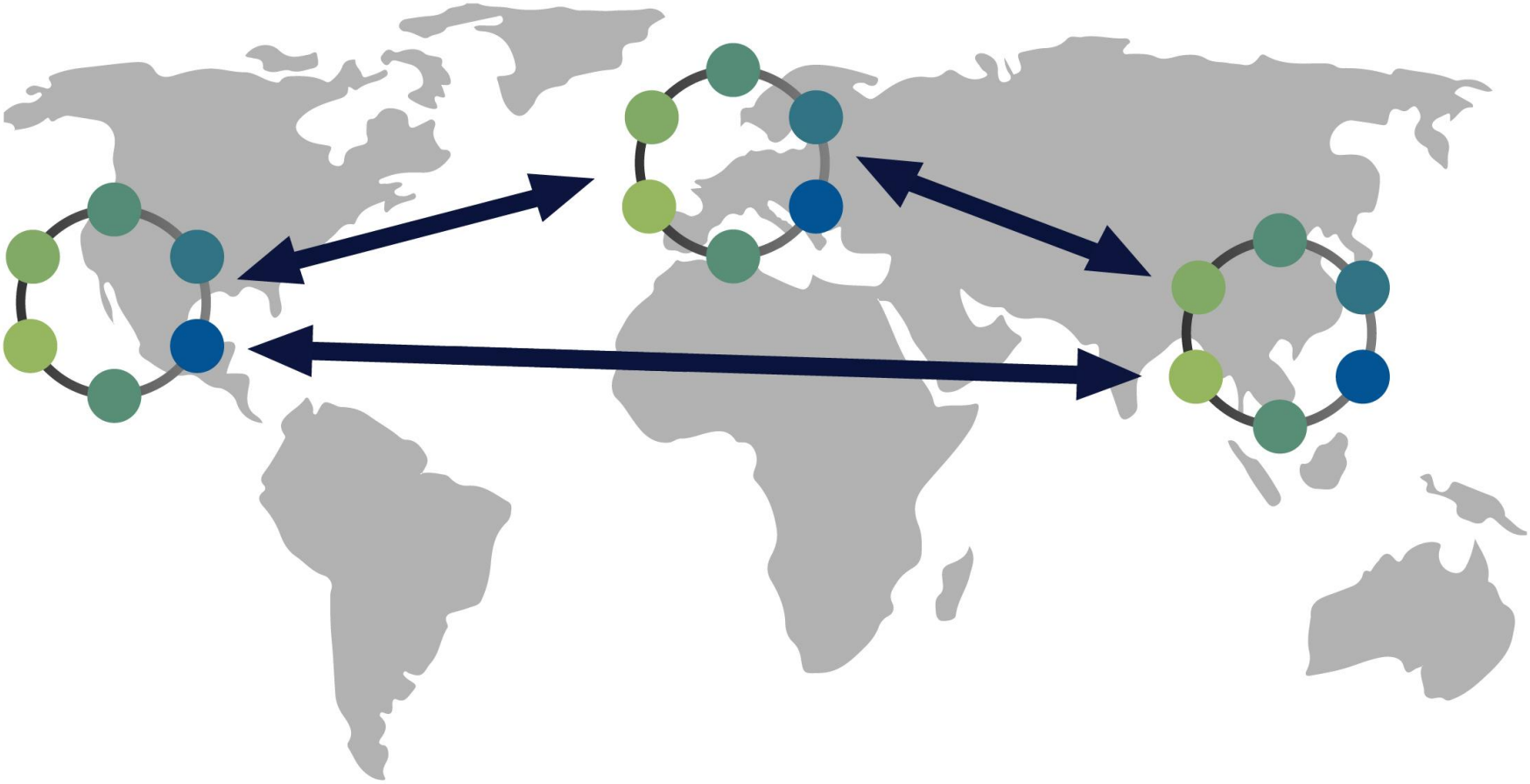
# Distributed vs Decentralised



# Distributed vs Decentralised



# Distributed vs Decentralised



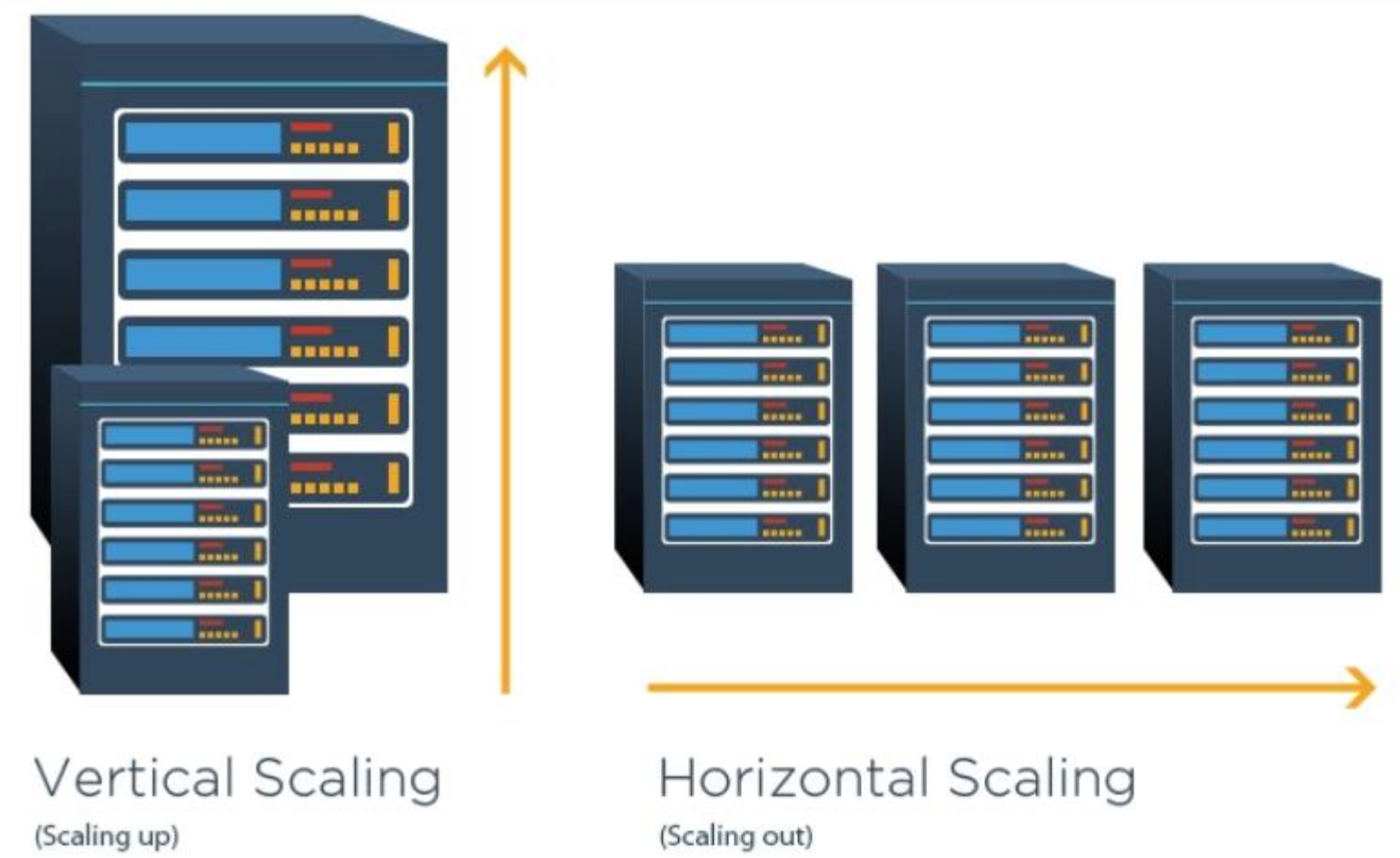
# Distributed vs Decentralised

- Decentralization, therefore, has **two key advantages**: it's simpler to use and avoids outages.
- It is simpler to operate and maintain a decentralised because **all nodes are the same.**
- It is not required special knowledge to scale; setting up 50 nodes isn't much different from setting up one.

# Elastic Scalability

- Scalability is an architectural feature of a system that can **continue serving a greater number of requests with little degradation in performance**.
- **Vertical scaling adds** more processing capacity and memory to your existing machine.
- **Horizontal scaling** means adding more machines that have all or some of the data on them so that no one machine has to bear the entire burden of serving requests.

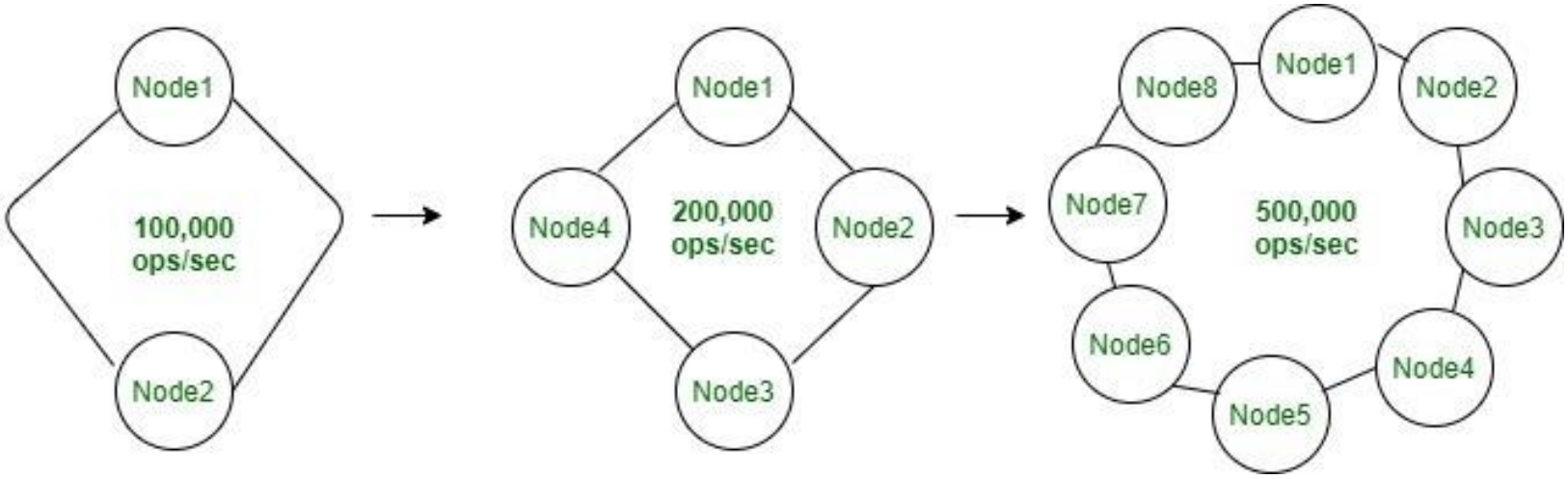
# Elastic Scalability



# Elastic Scalability

- **Elastic scalability** refers to a **special property** of **horizontal scalability**. It means that your cluster can seamlessly scale up and scale back down.
- The **cluster must be able to accept new nodes** that can begin participating by getting a copy of some or all of the data and **start serving new user requests without major disruption or reconfiguration** of the entire cluster.
- Its **not necessary to restart** the process, change the application queries or manually rebalance the data. Just add another machine.
- **Scaling down**, of course, **means removing some of the processing capacity** from the cluster. Example: seasonal workloads in retail or travel applications.

# Elastic Scalability



# High Availability and Fault Tolerance

- The **availability of a system** is measured according to its **ability to fulfill requests at any time**.
- Computers can experience different **failures**, from **hardware component** failure to **network disruption** to corruption.
- To mitigate failures, clusters include internal **hardware redundancies** and facilities **to send notification of failure events** and hot swap components.

# High Availability and Fault Tolerance

- A system to **be highly available**, it must typically include multiple networked computers, and the software they're running must then be capable of operating in a cluster and have some **facility for recognizing node failures** and failing over requests to another part of the system.

- **Cassandra is highly available**. The failed nodes can be replaced in the cluster with no downtime

- It is possible to **replicate data to multiple data centers** to offer improved local performance and prevent downtime if one data center experiences a catastrophe such as fire or flood.

# Tuneable Consistency

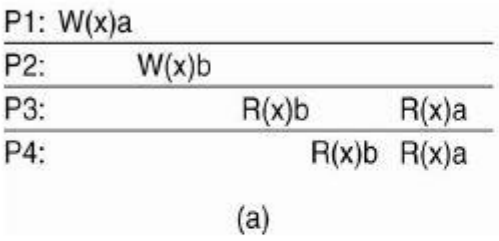
- Consistency: All the nodes should have same data at the same time
- **Consistency** is an overloaded term in the database world, but for our purposes we will use the definition that a read always **returns the most recently written value**.
- Cassandra trades some consistency in order to achieve total availability.
- With Cassandra the more accurately term is “tuneably consistent,” once it allows to easily decide the level of consistency you require, in balance with the level of availability.
- Eventual consistency is one of several consistency models available to architects. Let’s take a look at these models so we can understand the trade-offs:

# Tuneable Consistency

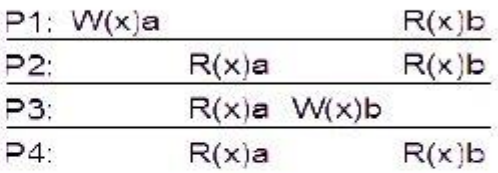
- **Strict consistency or sequential consistency** returns the most recently written value.

- In one single-processor machine, this is no problem to observe, as the sequence of operations is known to the one clock. But in a system executing across a variety of geographically dispersed data centers, it becomes much more slippery.

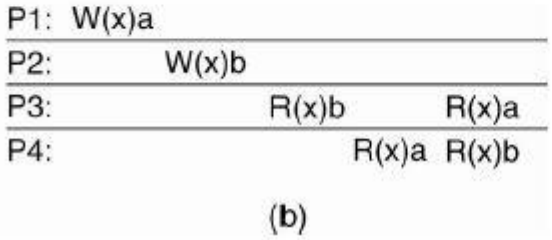
Three examples: (a) and (c) are a sequentially consistent data store. (b) a data store that is not sequentially consistent.




(a)




(c)




(b)

# Tuneable Consistency

- **Causal consistency** is a slightly weaker form of strict consistency. It does away with the fantasy of the single global clock that can magically synchronize all operations without creating an unbearable bottleneck.

- If two different, unrelated operations suddenly write to the same field at the same time, then those writes are inferred not to be causally related. But if one write occurs after another, we might infer that they are causally related.

**Example 1**: causal consistency

- This sequence is allowed with a causally-consistent store, but not with a sequentially consistent store.

| | | | | | |
|---|---|---|---|---|---|
| P1: | W(x)a | | | W(x)c | |
| P2: | | R(x)a | W(x)b | | |
| P3: | | R(x)a | | R(x)c | R(x)b |
| P4: | | R(x)a | | R(x)b | R(x)c |

**NOTE:** The writes W2(x)b and W1(x)c are concurrent, so it is not required that all processes see them in the same order.

# Tuneable Consistency

- **Weak (eventual) consistency** means on the surface that all updates will **propagate throughout all of the replicas** in a distributed system, but that this may take some time. Eventually, all replicas will be consistent.

- In Cassandra, consistency is not an all-or-nothing proposition. A more accurate term is ”**tuneable consistency**” because the **client can control the number** of replicas to block on for all updates. This is done by setting the consistency level against the replication factor.

# Brewer's CAP Theorem

- The theorem states that within a large-scale distributed data system, there are three requirements that have a relationship of sliding dependency:
- **Consistency** – All database clients will read the same value for the same query, even given concurrent updates.
- **Availability** – All database clients will always be able to read and write data.
- **Partition tolerance** – The database can be split into multiple machines; it can continue functioning in the face of network segmentation breaks.

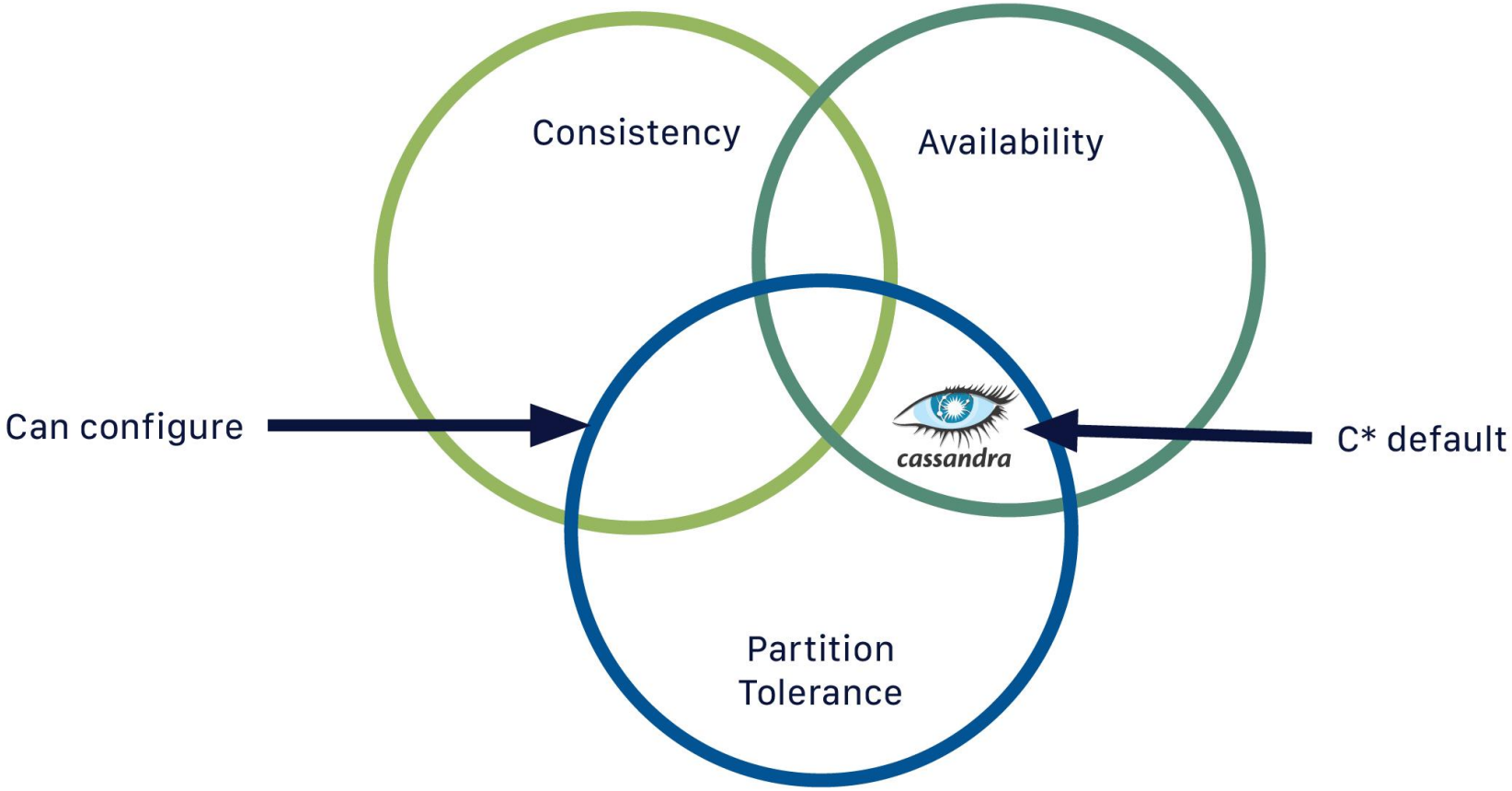
# Brewer's CAP Theorem

- Distributed system or data store can simultaneously provide only **two of three guarantees**: consistency, **availability, and partition tolerance** (CAP).

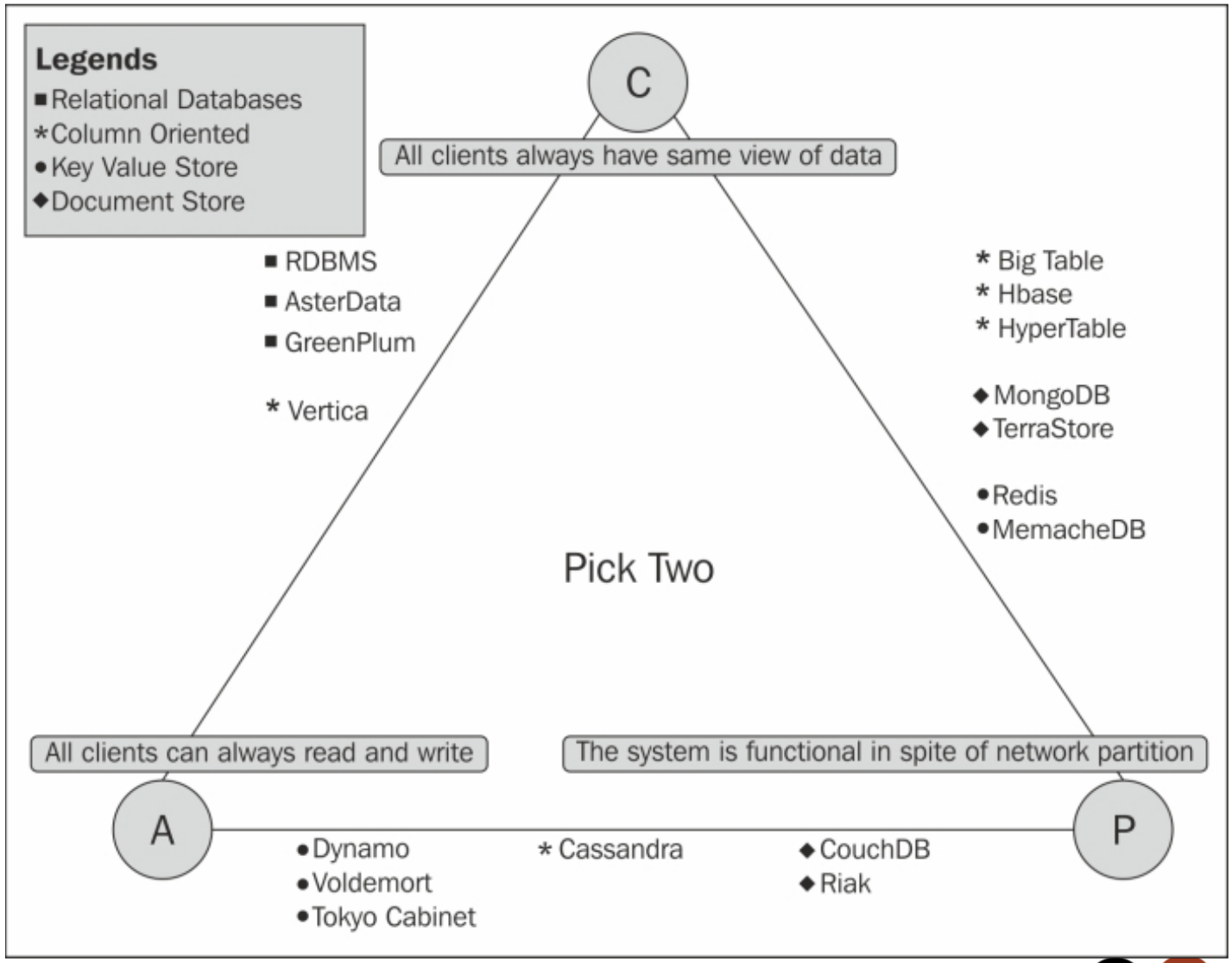
- This is analogous to the saying you may have heard in software development:

"You can have it good, you can have it fast, you can have it cheap: pick two."

# Brewer's CAP Theorem



# Brewer's CAP Theorem



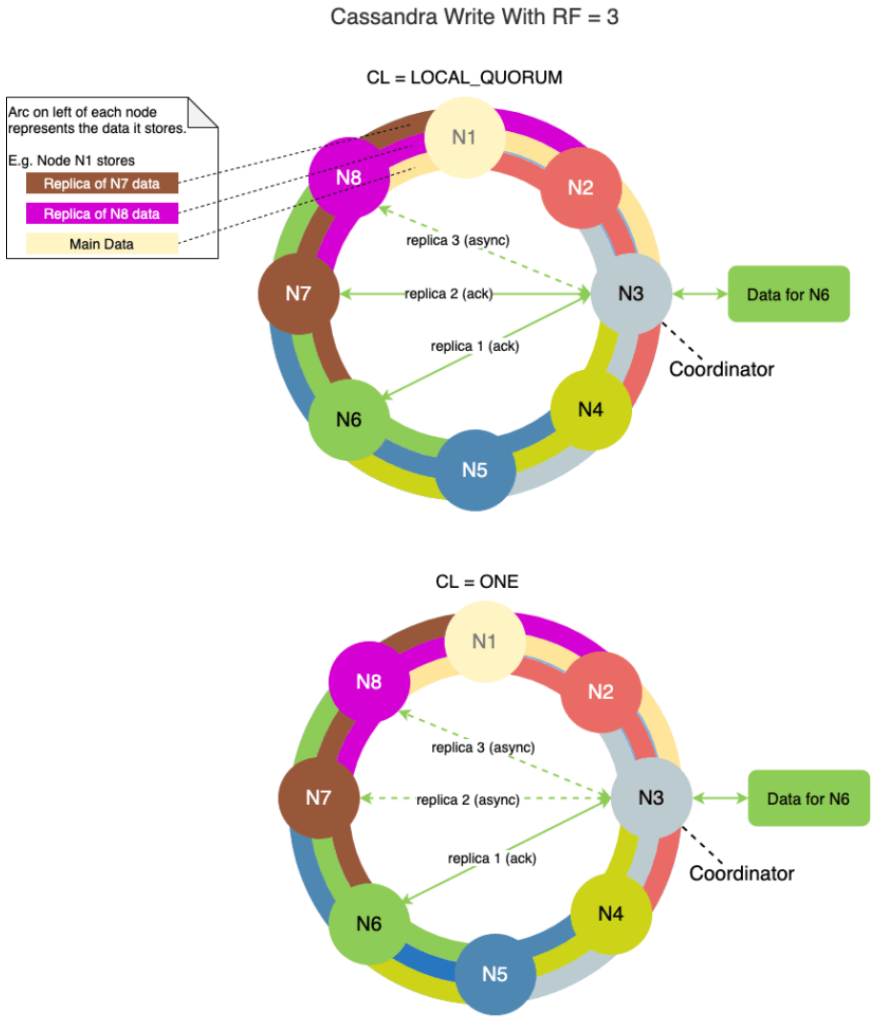
# Data Replication

- To achieve **high availability**, Cassandra relies on the **replication** of data across clusters.
- **Replication Factor** (RF) specifies **how many nodes across** the cluster would store the replicas
- Two strategies:
  - **SimpleStrategy** is used for a single data center and one rack topology
  - **NetworkTopologyStrategy** is generally used for multiple datacenters and multiple racks.

# Consistency Level

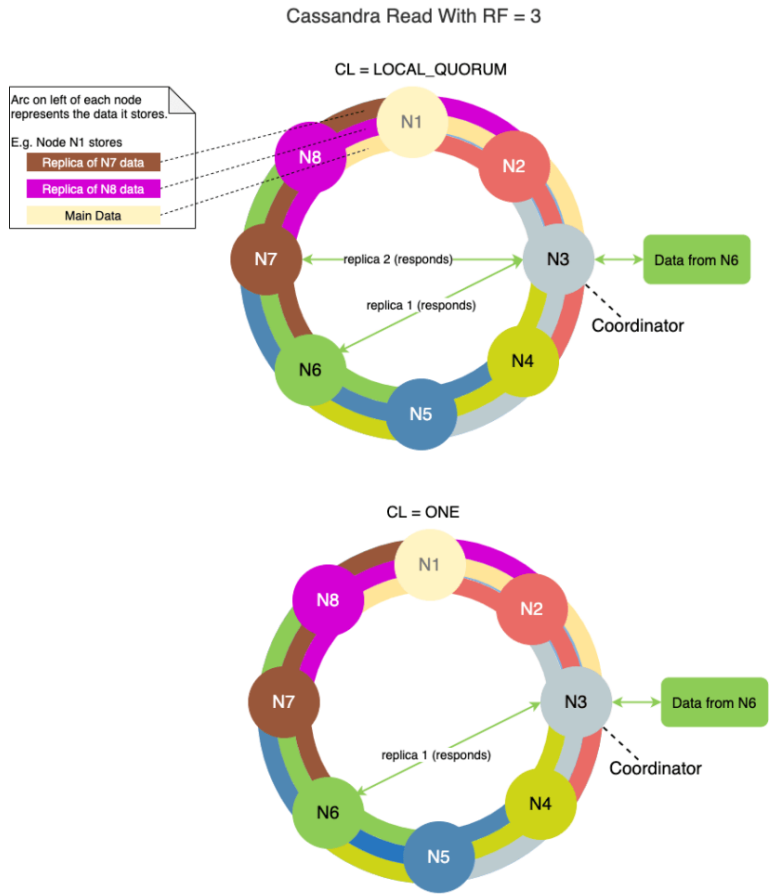
- Consistency level ONE means it needs acknowledgment from only one replica node
- Consistency level QUORUM means it needs acknowledgment from 51% or a majority of replica nodes across all datacenters.
- Consistency level of LOCAL_QUORUM means it needs acknowledgment from 51% or a majority of replica nodes just within the same datacenter as the coordinator.
- Consistency level of ALL means it needs acknowledgment from all the replica nodes

# Consistency Level



# Consistency Level on Read

- For read operations, the consistency level specifies how many replica nodes must respond with the latest consistent data before the coordinator successfully sends the data back to the client.



# Queries and Coordinator Nodes

- A **client may connect to any node** in the cluster to initiate a read or write query. This node is known as the **coordinator node.**
- The **coordinator identifies which nodes are replicas** for the data that is being written or read and forwards the queries to them.
- **For a write, the coordinator node contacts all replicas**, as determined by the **consistency level** and replication factor, and considers the write successful when a number of replicas commensurate with the consistency level acknowledge the write.
- For a read, the **coordinator contacts enough replicas** to ensure the required **consistency level** is met, and returns the data to the client.

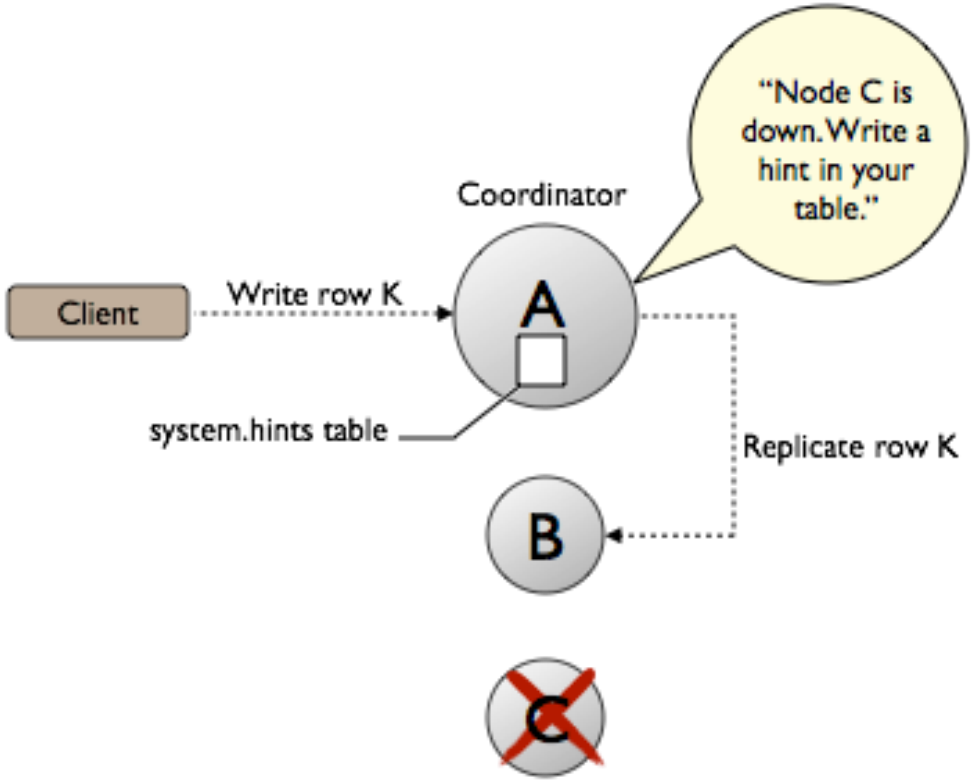
# Hinted Handoff

- Cassandra feature that optimizes the cluster consistency process when a replica node is not available, due to network issues or other problems

- Coordinator will create a hint, which is a small reminder
- In a cluster of three nodes, A (the coordinator), B, and C, each row is stored on two nodes in a keyspace having a replication factor of 2. Suppose node C goes down. The client writes row K to node A. The coordinator, replicates row K to node B, and writes the hint for downed node C to node A.

- When node C comes back up, node A reacts to the hint by forwarding the data to node C.

# Hinted Handoff



# Memtables, SSTables, and Commit Logs

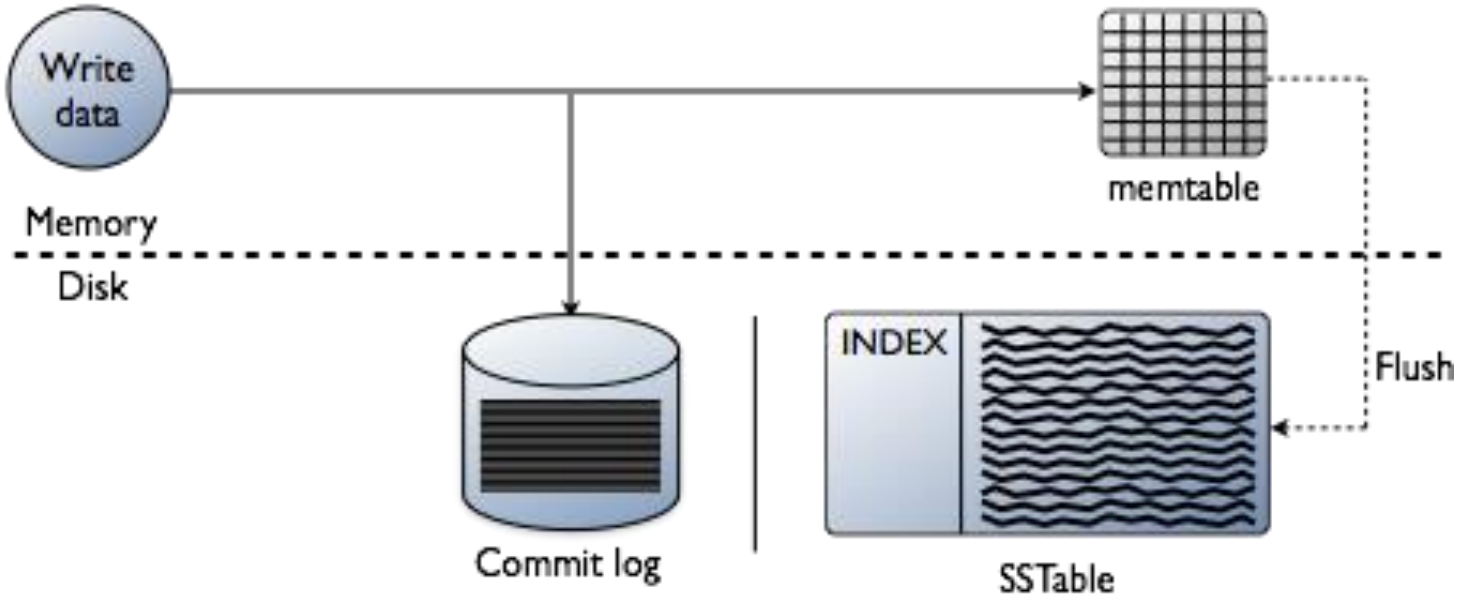
- When a node receives a write operation, it immediately writes the

data to a commit log. The commit log is a crash-recovery mechanism that supports Cassandra's durability goals.

- When the number of objects stored in the memtable reaches a threshold, the contents of the memtable are flushed to disk in a file called an SSTable.

- Cassandra will read both SSTables and memtables to find data values, as the memtable may contain values that have not yet been flushed to disk.

# Memtables, SSTables, and Commit Logs



# Installation of Apache Cassandra

# Installation of Apache Cassandra

- Installation of the Java Development Kit 1.8 – update 251
- Environment variables Definition
  - set JAVA_HOME = path of JDK
- Installation of Python 2.7
- Environment variables Definition
  - Path = path of Python2.7
- MacOS (ensure that you have the versions of JDK and python 2.7 in your mac)
  - Install brew
  - brew update
  - brew install cassandra

# Running Apache Cassandra in Python

- Create a new envirement in Anaconda for python 2.7

  - `conda create -n cass_env python=2.7`
  - `conda activate cass_env`
  - `conda install pip`
  - `pip install pandas`
  - `pip install futures`
  - `pip install cassandra-driver`
  - `pip install kafka`

# Installation of Apache Cassandra

- Download Apache Cassandra 3.11

- Installation Folder – Copy the Apache Cassandra unzip folder to
  - ”C:/Cassandra/”

- Environment variables Definition
  - CASSANDRA_HOME = path of Cassandra

- Open two terminals for the server and client
- Put both in the directory `cd %CASSANDRA_HOME%/bin`

- To run the server type `cassandra`
- To run the client type `cqlsh`

Check the version of java and python

-- java version "1.8.0_251“
-- Python 2.7.18

# Cassandra Example

- In the client use the following commands:

```
create keyspace course with replication =
{'class':'SimpleStrategy','replication_factor':1};

describe keyspaces;

use courses;

create table student (id int primary key, name text);

select * from student;
```

- What are we creating?

# Running Apache Cassandra in Python

```
from cassandra.cluster import Cluster


if __name__ == "__main__" :
    cluster = Cluster(['127.0.0.1'] , port =9042)
    session = cluster.connect( 'course', wait_for_all_pools =True )
    session.execute('USE course')
    session.execute ('insert into student (id, name) values (11
,\'Fatima\'); ')
    rows = session.execute ( "SELECT * FROM student ")
    for row in rows :
        print (row.id, row.name)
```

# Exercises

- Create a name space and a set of tables to store a shopping cart and respective clients and products tables. Use the CQLSH client.
- Create a name space and a set of tables to store medical data like patient name, age, address and the doctor name. Use the CQLSH client.
- Create a name space and set a table to store the boston dataset. Use the python language to insert and select the elements.

# Relational Data Model

- Database itself is the outermost container that might correspond to a single application.
- The database contains tables. Tables have names and contain one or more columns, which also have names.
- When you add data to a table, you specify a value for every column defined; if you don’t have a value for a particular column, you use null.

# Relational Data Model

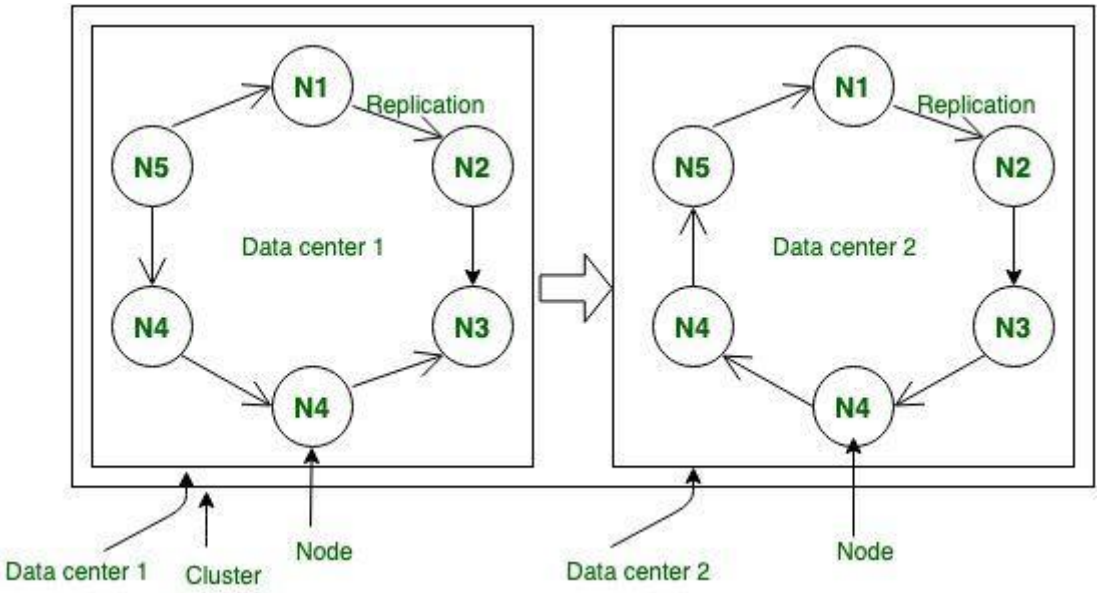
- This new entry adds a row to the table, which you can later read if you know the row's unique identifier (primary key), or by using a SQL statement that expresses some criteria that row might meet.

- If you want to update values in the table, you can update all of the rows or just some of them, depending on the filter you use in a "where" clause of your SQL statement.

# Architecture of Cassandra

# Architecture of Cassandra

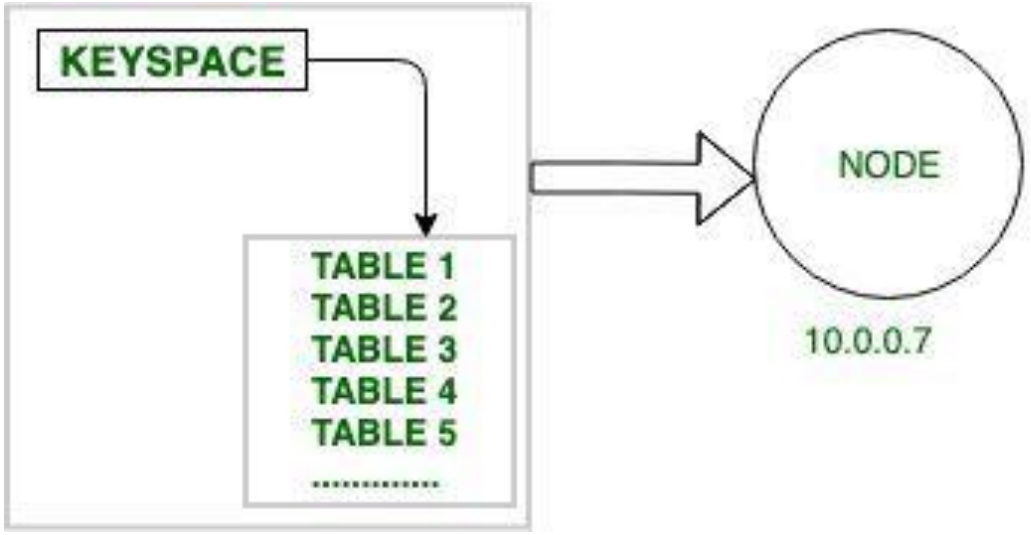
- Basic terminology:
  - Node
  - Data center
  - Cluster
- Operations
  - Read
  - Write
- Data Replication strategies
  - Simple Strategy
  - Network Topology Strategy



# Architecture of Cassandra

- Node is the basic component in Apache Cassandra. It is the place where actually data is stored.

- keyspace contains one or more tables



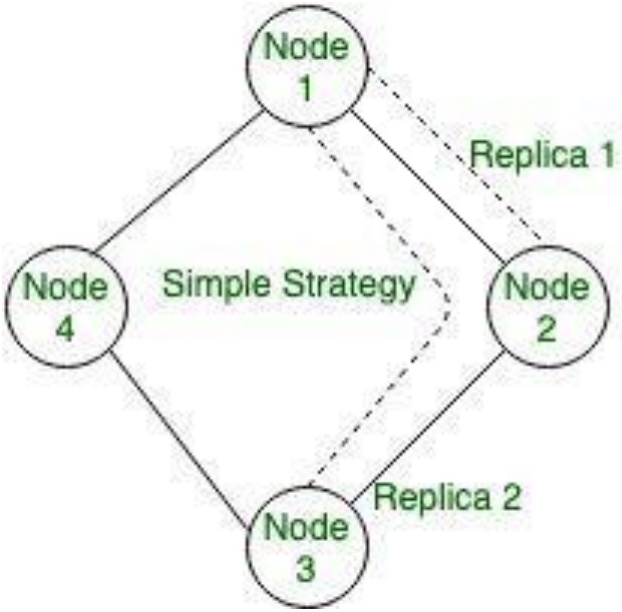
# Example

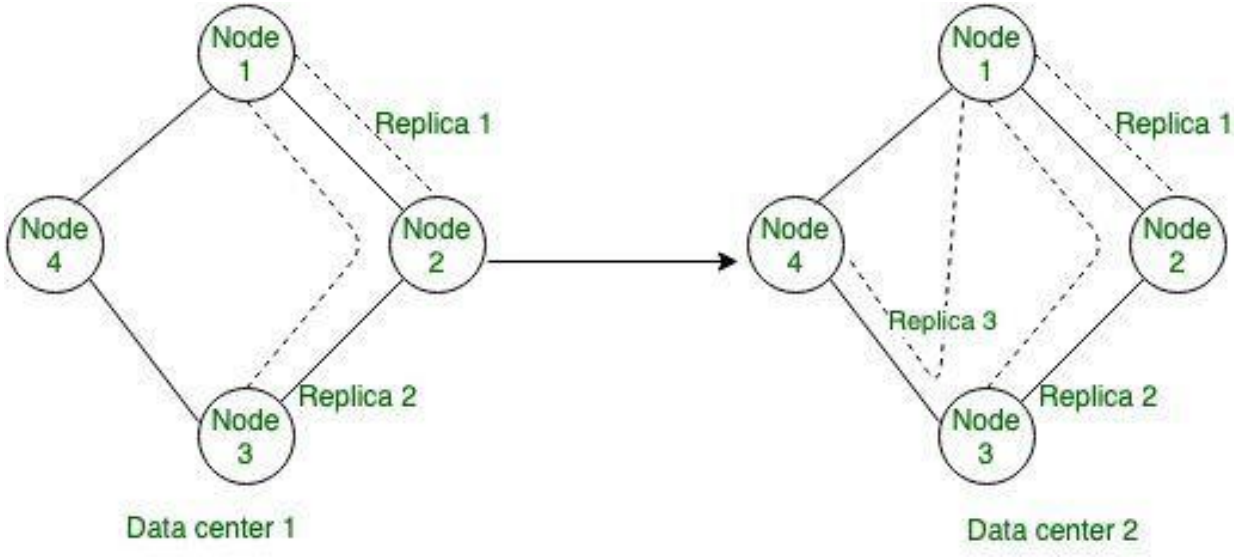
```
CREATE KEYSPACE User_data
    WITH replication = {'class': 'SimpleStrategy',
                        'replication_factor' : 2};
```

```
CREATE KEYSPACE User_data
    WITH replication = {'class': 'NetworkTopologyStrategy', 'DC1' : 2, 'DC2' :
3}
    AND durable_writes = false;
```

```
DESCRIBE KEYSPACE User_data
```

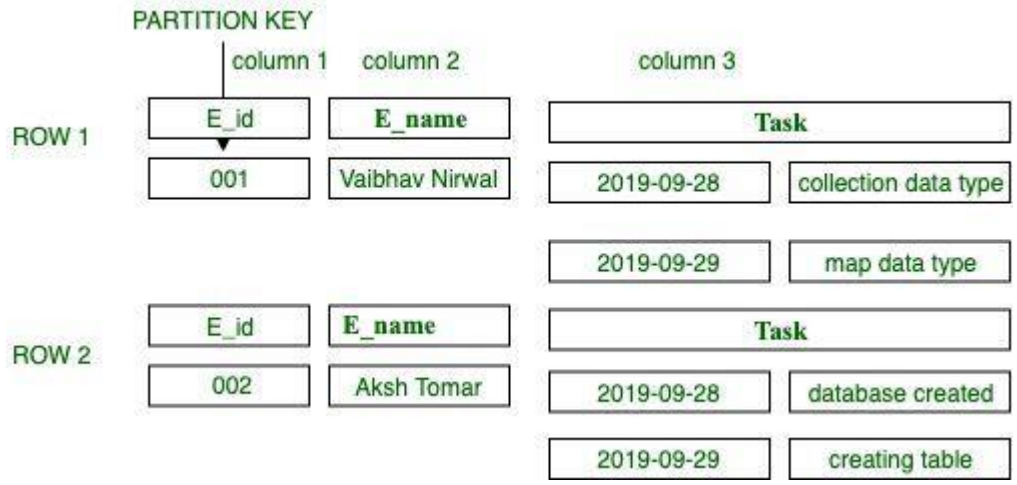
# Cassandra Example



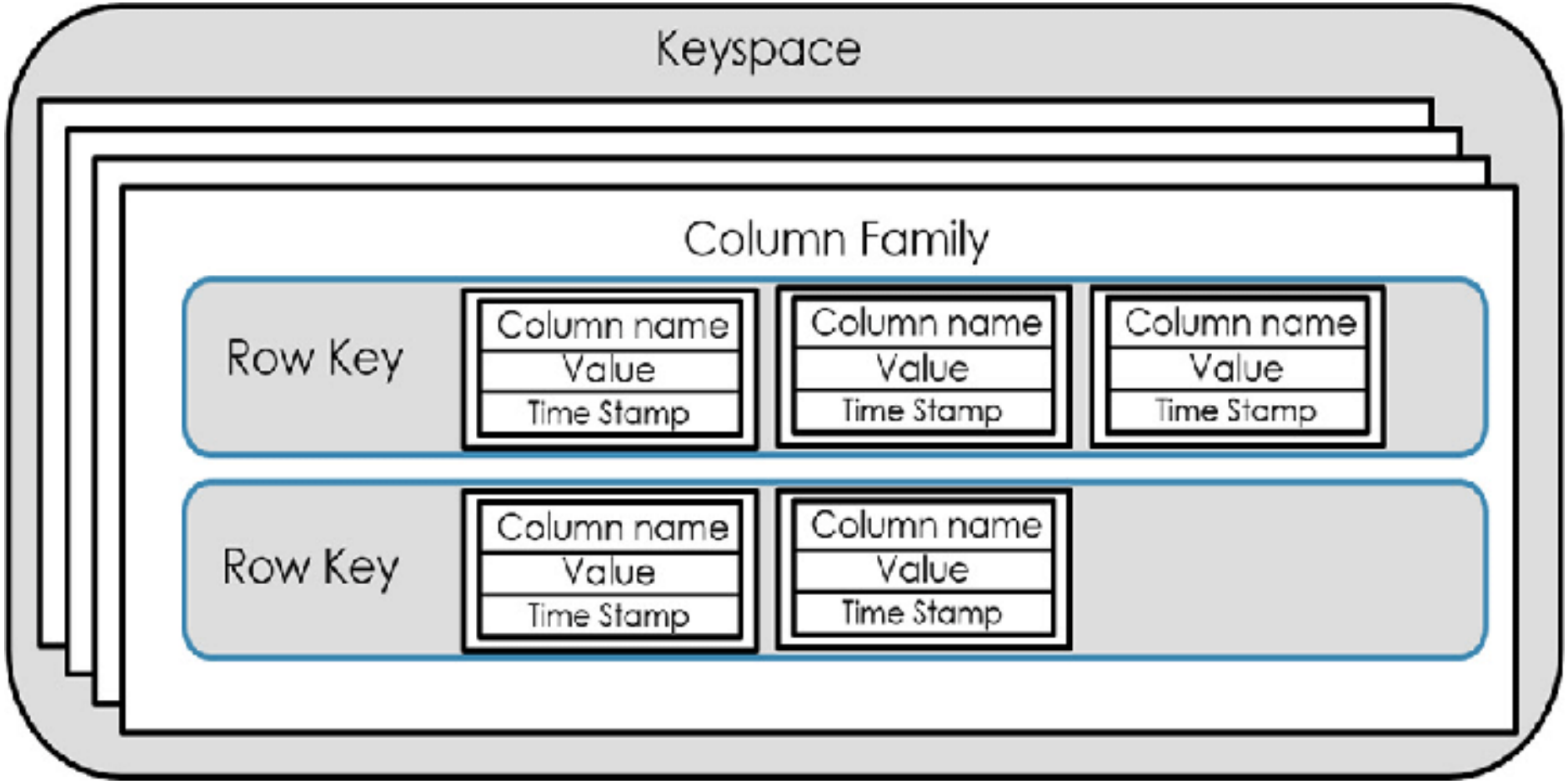



# Cassandra Data Model

- The simplest data store you would conceivably want to work with might be an array or list. An array is a clearly useful data structure, but not semantically rich.
- Now let's add a second dimension to this list: names to match the values. This is an improvement because you can know the names of your values.
- To store multiple entities with the same structure we need rows



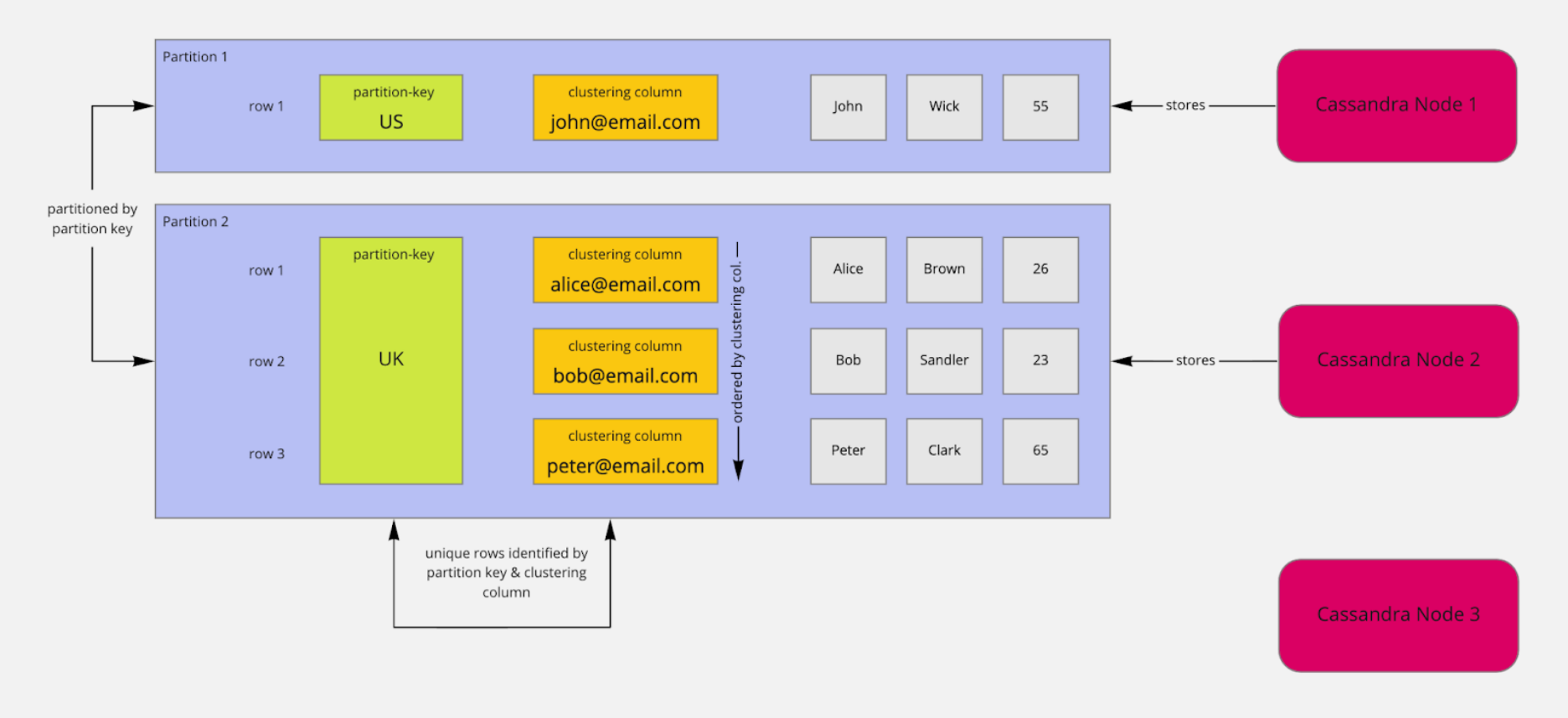
# Cassandra Data Model



# Cassandra Data Model

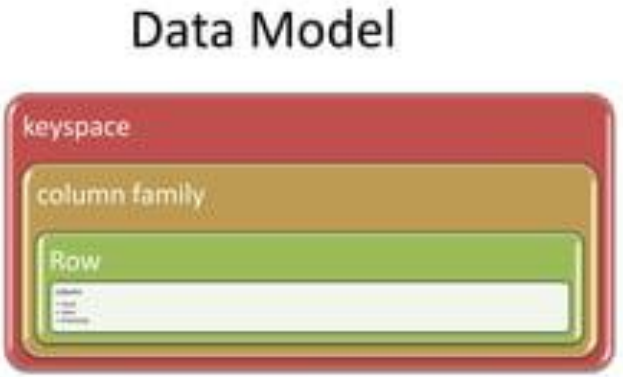
- Name/value pairs columns
- Unique identifier for each row could be called a row key or primary key
- Cassandra uses a special type of primary key called a composite key (or compound key) to represent groups of related rows, also called partitions
- The composite key consists of a partition key, plus an optional set of clustering columns
- The partition key determines the nodes on which rows are stored
- Clustering columns are used to control how data is sorted for storage within a partition.

# Cassandra Data Model



# Cassandra Data Model

- We have the basic Cassandra data structures:
  - **Column**, which is a name/value pair
  - **Row**, which is a container for columns referenced by a primary key
  - **Partition**, which is a group of related rows that are stored together on the same nodes
  - **Table**, which is a container for rows organized by partitions
  - **Keyspace**, which is a container for tables
  - **Cluster**, which is a container for keyspaces that spans one or more nodes




| Cassandra | RDBMS Equivalent |
|---|---|
| KEYSPACE | DATABASE/SCHEMA |
| COLUMN FAMILY | TABLE |
| ROW | ROW |
| FLEXIBLE COLUMNS | DEFINED COLUMNS |

# Cassandra Data Model

- Each time you write data into Cassandra, a **timestamp,** in microseconds, is generated for each column value that is inserted or updated.
- Cassandra uses these timestamps for **resolving any conflicting** changes that are made to the same value, in what is frequently referred to as a last write wins approach.
- One very powerful feature that Cassandra provides is the ability to **expire data that is no longer needed**. This expiration is very flexible and works at the level of individual column values.
- The **time to live** (or TTL) is a value that Cassandra stores for each column value to indicate how long to keep the value.

# Cassandra Query Language (CQL)

- CQL supports a flexible set of data types, including simple character and numeric types, collections, and user-defined types.
- CQL supports the numeric types you'd expect, including integer and floating-point numbers. These types are similar to standard types in Java and other languages: int, bigint, smallint, tinyint, varint, float, double, decimal
- CQL provides two data types for representing text, one of which you've made quite a bit of use of already (text): text, varchar, ascii
- https://docs.datastax.com/en/cql-oss/3.3/cql/cqlIntro.html

# Cassandra Query Language (CQL)

- The identity of data elements such as rows and partitions is important in any data model in order to be able to access the data. Cassandra provides several types that prove quite useful in defining unique partition keys: timestamp, date, time, uuid, timeuuid

- CQL provides several other simple data types that don't fall nicely into one of the preceding categories: boolean, blob, inet, counter

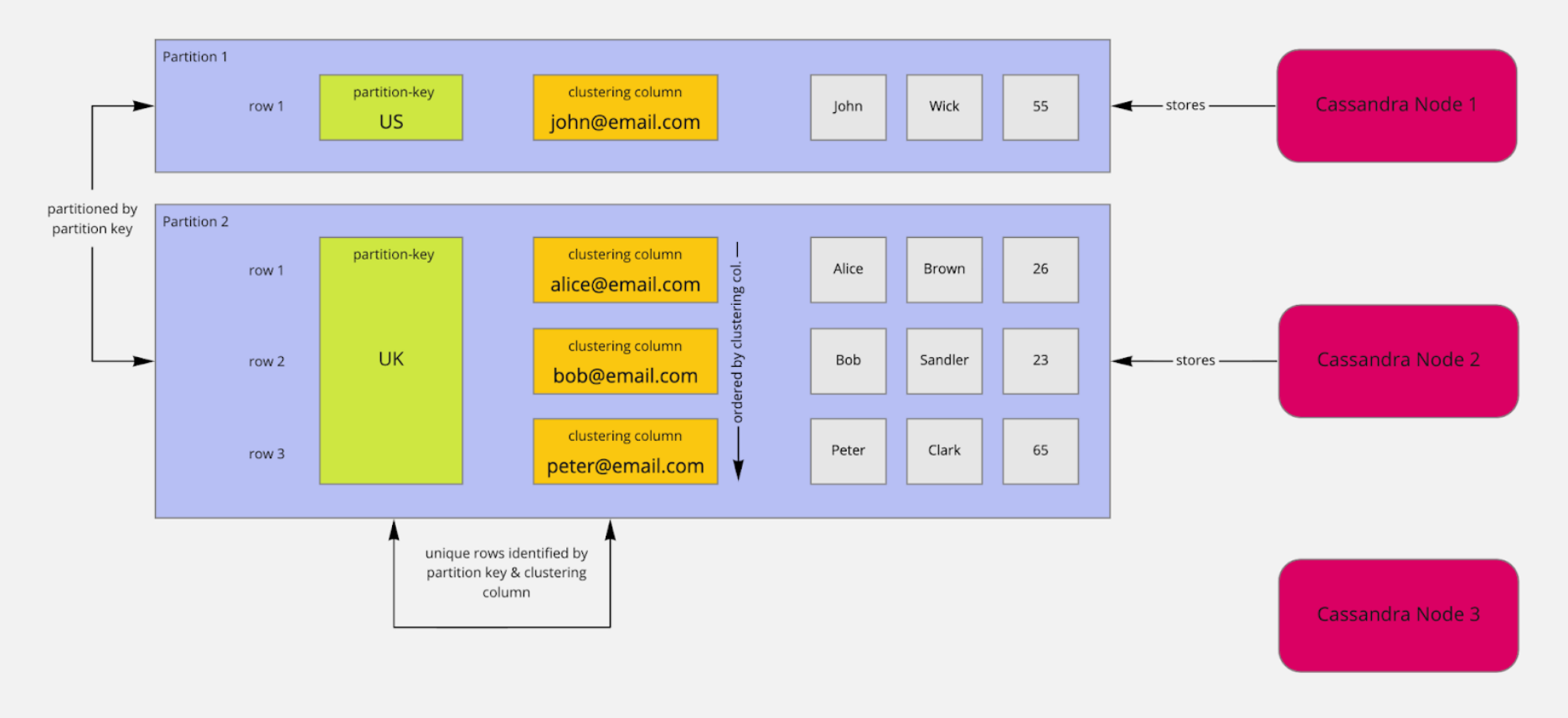
- CQL provides three collection types to help you with these situations: set, list, and map

- Cassandra provides two different ways to manage more complex data structures: tuples and user-defined types.

# Cassandra example

```
CREATE TABLE learn_cassandra.users_by_country (
    country text,
    user_email text,
    first_name text,
    last_name text,
    age smallint,
    PRIMARY KEY ((country), user_email)
)
```

# Cassandra Data Model



# Cassandra Connector

# Cassandra Connector

- Create the following table in cassandra

```
create keyspace my_ks with replication =
{'class':'SimpleStrategy','replication_factor':1};

describe keyspaces;

use my_ks;

create table my_tbl (myid int primary key, values text);

select * from my_tbl;
```

# Cassandra Connector

- Download Cassandra Connector
- Create ”plugin” folder inside Kafka folder
- Copy the Cassandra Connector to ”plugin” folder inside Kafka folder
- Copy the jar file kafka-connect-cassandra-sink-1.4.0 of the plugin to the libs folder of kafka
- Copy the cassandra-sink-standalone.properties.example to Kafka/config
- Change the name to cassandra-sink-standalone.properties

# Cassandra Connector

- Edit the file and set the ”Topic” and ”Kafka record field to database”
- Edit to ”topic.my_topic.my_ks.my_tbl.mapping=myid=key, values=value”
- Notice:
  - my_topic is the name of the topic to be created in the kafka producer
  - my_ks is the keyspace name of Cassandra
  - my_tbl is the name of the table in Cassandra
  - myid is the primary key of the table
  - values are the values in the table
- These values should be configured according to the parameters of a table

# Cassandra Connector

- Change the remaining file accordingly:
  - topic.my_topic.my_ks.my_tbl.ttlTimeUnit
  - topic.my_topic.my_ks.my_tbl.timestampTimeUnit

- Edit the connect-standalone.properties and set the
  - ”key.converter.schemas.enable=false” and
  - ”value.converter.schemas.enable=false”

- Start kafka and zookeeper:
  - cd %KAFKA_HOME%/bin/windows
  - zookeeper-server-start.bat ../../config/zookeeper.properties
  - kafka-server-start.bat ../../config/server.properties

# Cassandra Connector

- Start the conector to receive values from kafka

- cd %KAFKA_HOME%/bin/
- connect-standalone.bat ../../config/connect-standalone.properties ../../config/cassandra-sink-standalone.properties

# Cassandra Connector

```
import json
import threading
from time import sleep
from kafka import KafkaProducer

def thread_function (name, sl):
    producer = KafkaProducer (bootstrap_servers =['localhost:9092'],
                              key_serializer = str.encode, value_serializer =
                              lambda v : json.dumps(v).encode('utf-8'))
    for e in range (10) :
        producer.send ('my_topic', key = str ( e ), value = 'bar' + str ( e ) )
        print('bar' + str ( e ))
        sleep (sl)

x1 = threading.Thread (target = thread_function , args =( "1" ,5,) )
x1.start ()
x1.join ()
```

# Exercises

- Create two producers that send in parallel random numbers with different time intervals. The connector should send the data to the datastore.

- Create N producers that send the information of each attribute of the boston dataset. The connector should send the data to the datastore.

# Work to be done to the next class

- Create a table to store important information from kafka producers that you have created for your dataset

- Finish in the paper:
  - Abstract
  - Introduction
  - Related-work

- 10th may submit in Moodle the paper and the python scripts

# References


- Estrada, R., & Ruiz, I. (2016). Big data smack. Apress, Berkeley, CA.
- Kumar, M., & Singh, C. (2017). Building Data Streaming Applications with Apache Kafka. Packt Publishing Ltd

UNIVERSIDADE
PORTUCALENSE

Do conhecimento à prática.