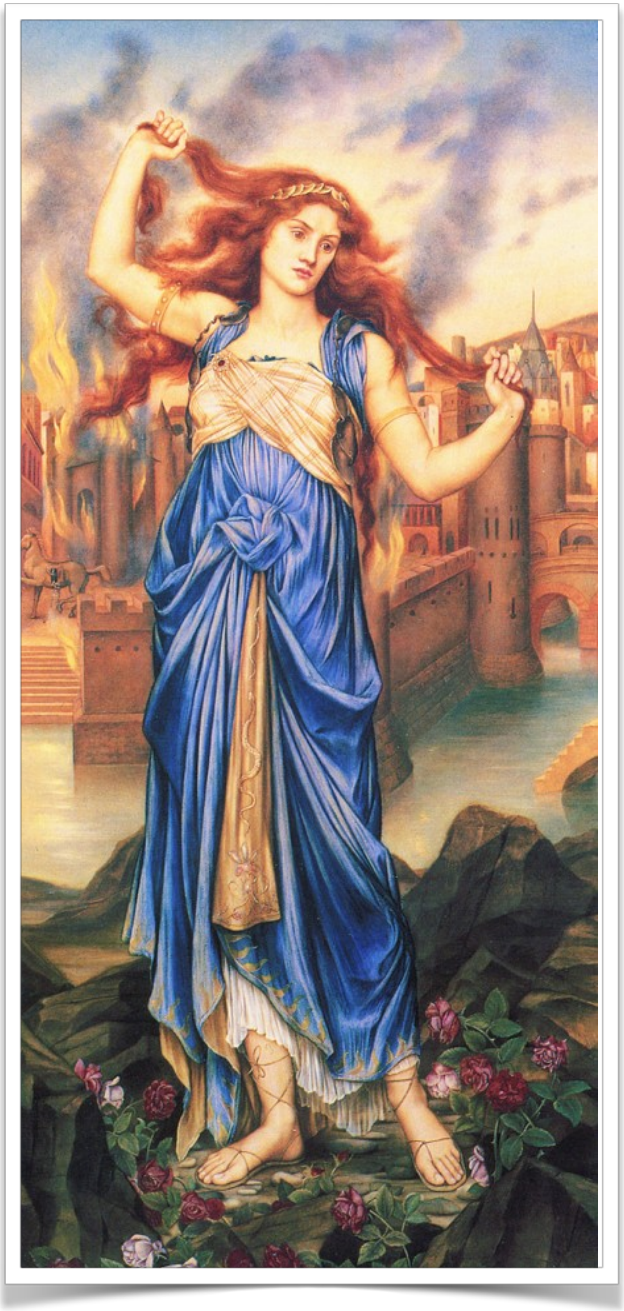
# CS595—BIG DATA TECHNOLOGIES

Module 13

NoSQL Databases: Cassandra

# CASSANDRA

- A.k.a Alexandra or Kassandra
- Daughter of King Priam and Queen Hecuba of Troy.
- Apollo gave her the power of prophecy to seduce her. She refused and then Apollo cursed her never to be believed.
- https://en.wikipedia.org/wiki/Cassandra



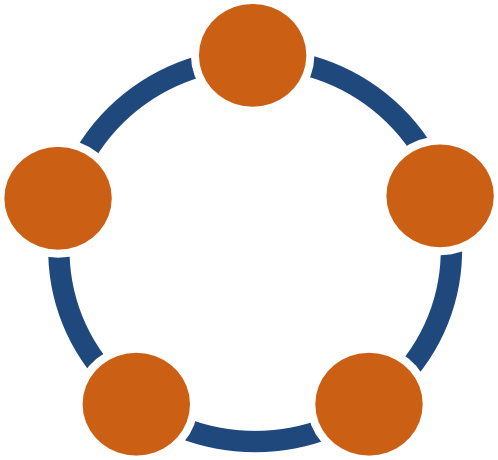
# CASSANDRA

- Open Source distributed database management system
- Initially developed at Facebook
- Inspired by Amazon's Dynamo and Google BigTable papers
- Became Apache top-level project in Feb, 2010
- Nowadays developed by DataStax

# Cassandra

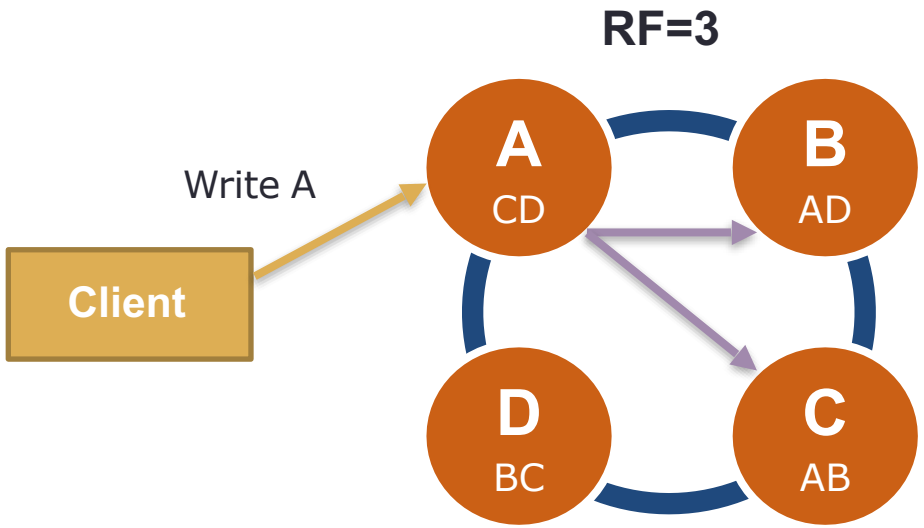
- A Linearly Scaling and Fault Tolerant Distributed Database

- Fully Distributed
  - Data spread over many nodes
  - All nodes participate in a cluster
  - All nodes are equal
  - No SPOF (shared nothing)



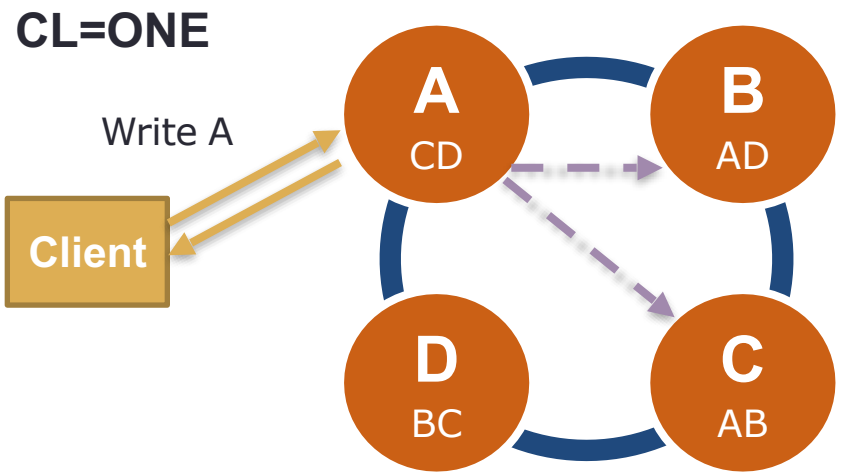
# Two knobs control Cassandra fault tolerance

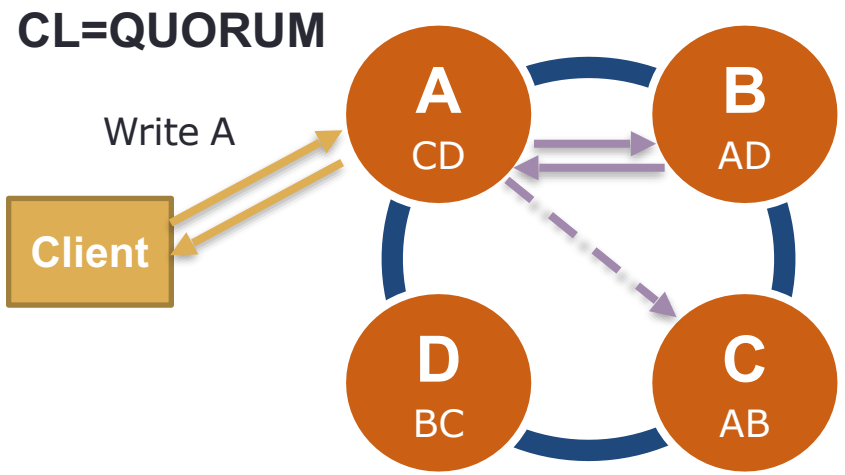
- Replication Factor (server side)
  - How many copies of the data should exist?



# Two knobs control Cassandra fault tolerance

- Consistency Level (client side)
  - How many replicas do we need to hear from before we acknowledge?



# Consistency Levels

- Applies to both Reads and Writes (i.e. is set on each query)

  **ONE** – one replica from any DC

  **LOCAL_ONE** – one replica from local DC

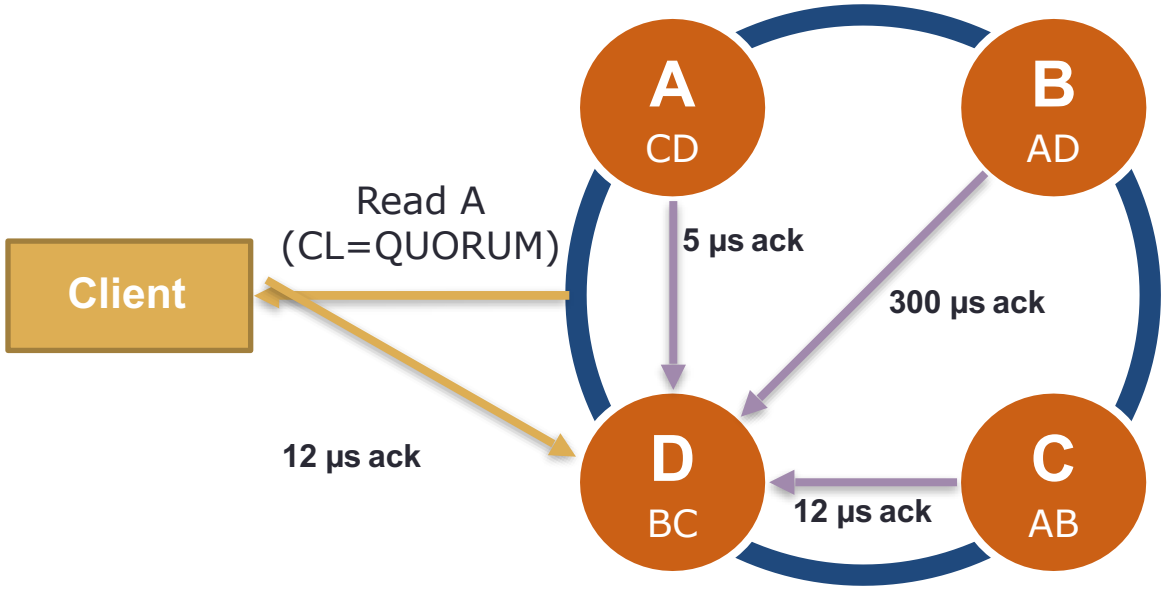
  **QUORUM** – 51% of replicas from any DC

  **LOCAL_QUORUM** – 51% of replicas from local DC

  **ALL** – all replicas

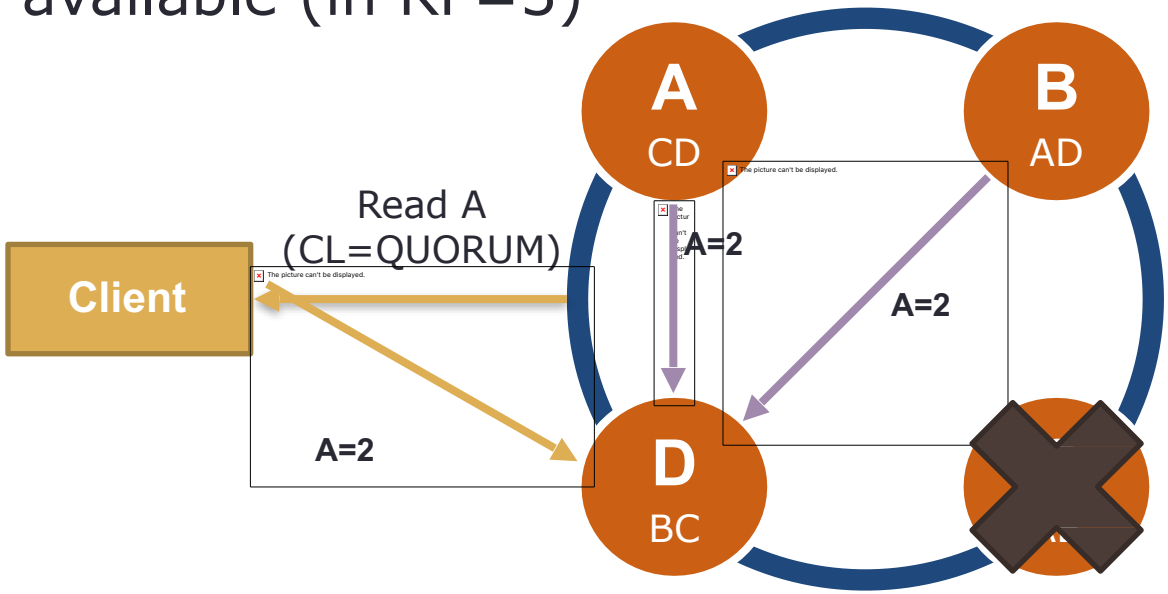
# Consistency Level and Speed

- How many replicas we need to hear from can affect how quickly we can read and write data in Cassandra



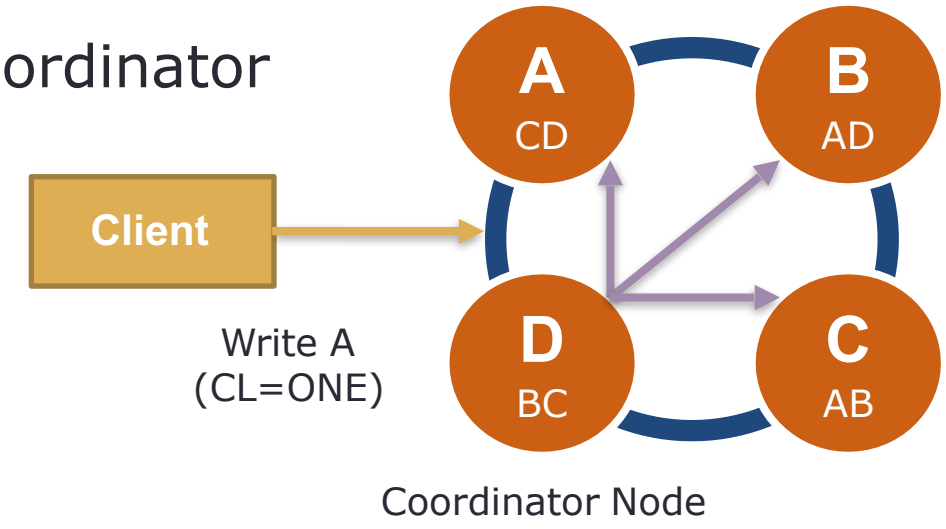
# Consistency Level and Availability

- Consistency Level choice affects availability
- For example, QUORUM can tolerate one replica being down and still be available (in RF=3)



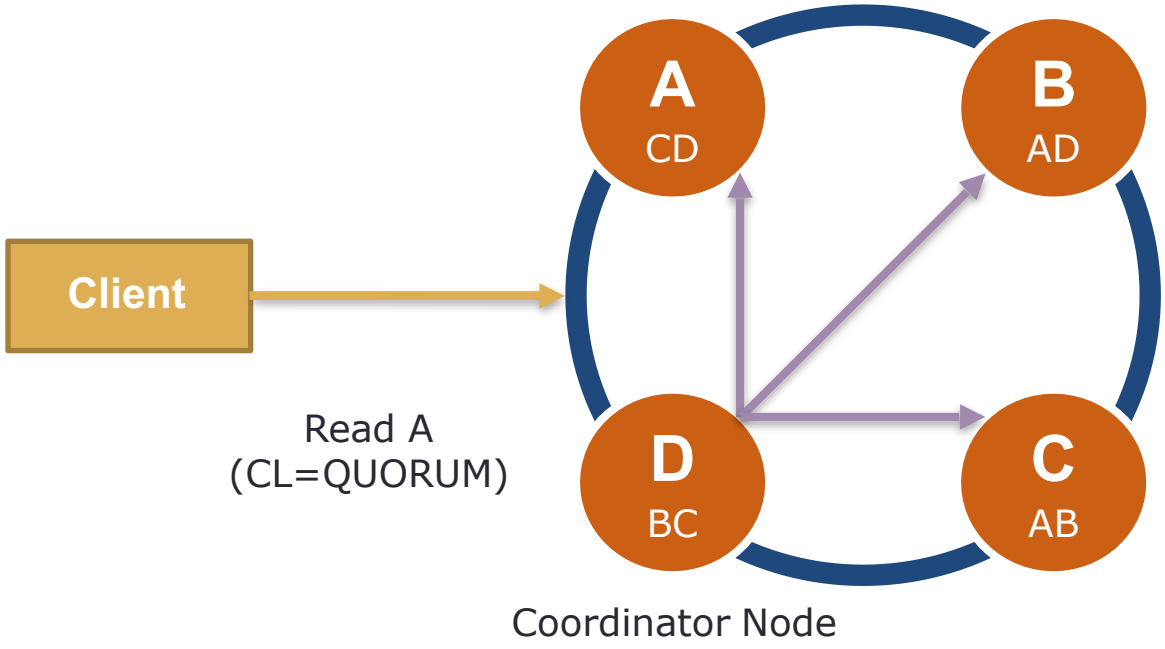
# Writes in the cluster

- Fully distributed, no SPOF
- Node that receives a request is the Coordinator for request
- Any node can act as Coordinator



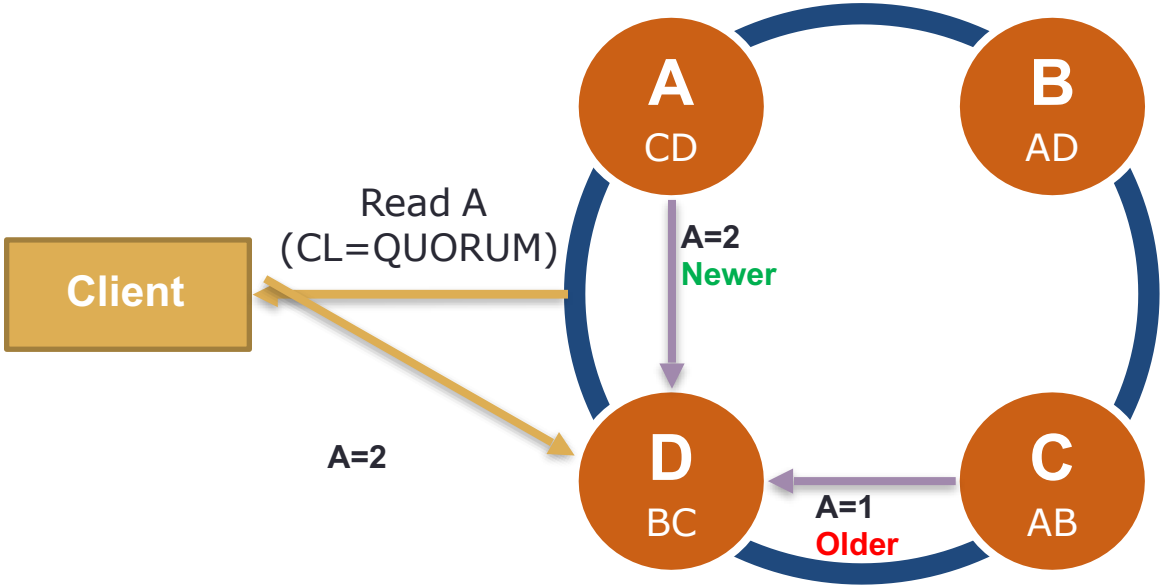
# Reads in the cluster

- Same as writes in the cluster, reads are coordinated
- Any node can be the Coordinator Node



# Reads and Eventual Consistency

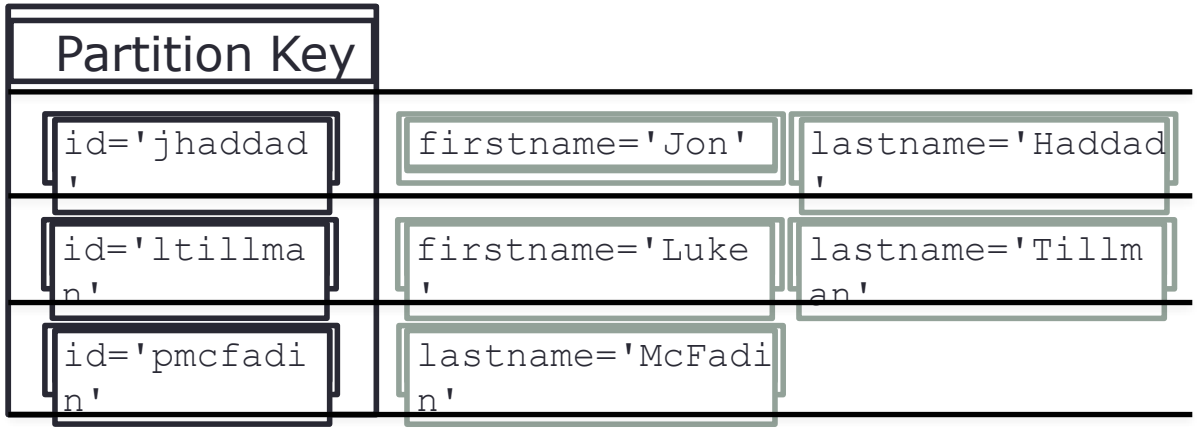
- Cassandra is an **AP** system that is **Eventually Consistent** so replicas may disagree
- Column values are timestamped
- In Cassandra, Last Write Wins (LWW)



# Data Distribution

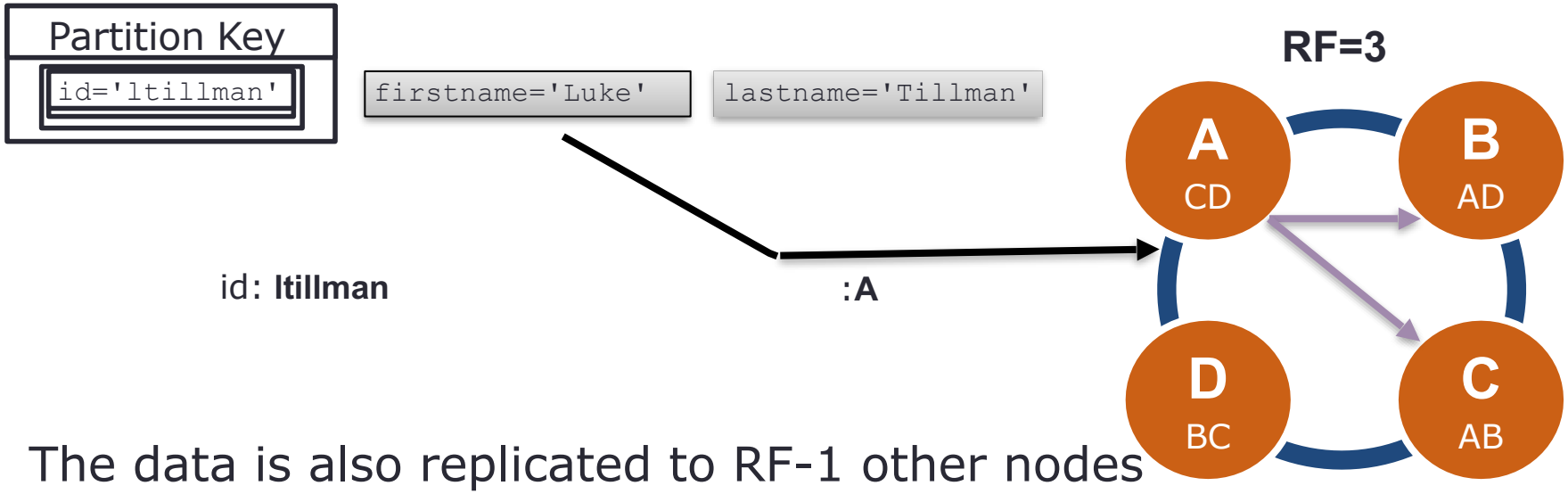
- Partition Key determines node placement

```
CREATE TABLE users (
    id text,
    firstname text,
    lastname text,
    PRIMARY KEY (id)
);
```



# Data Distribution

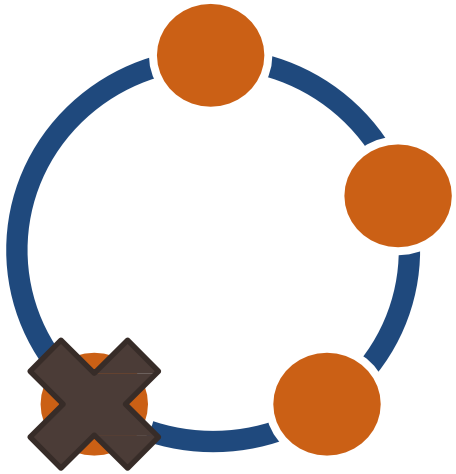
- The Partition Key is hashed using a consistent hashing function and the output is used to place the data on a node




- The data is also replicated to RF-1 other nodes

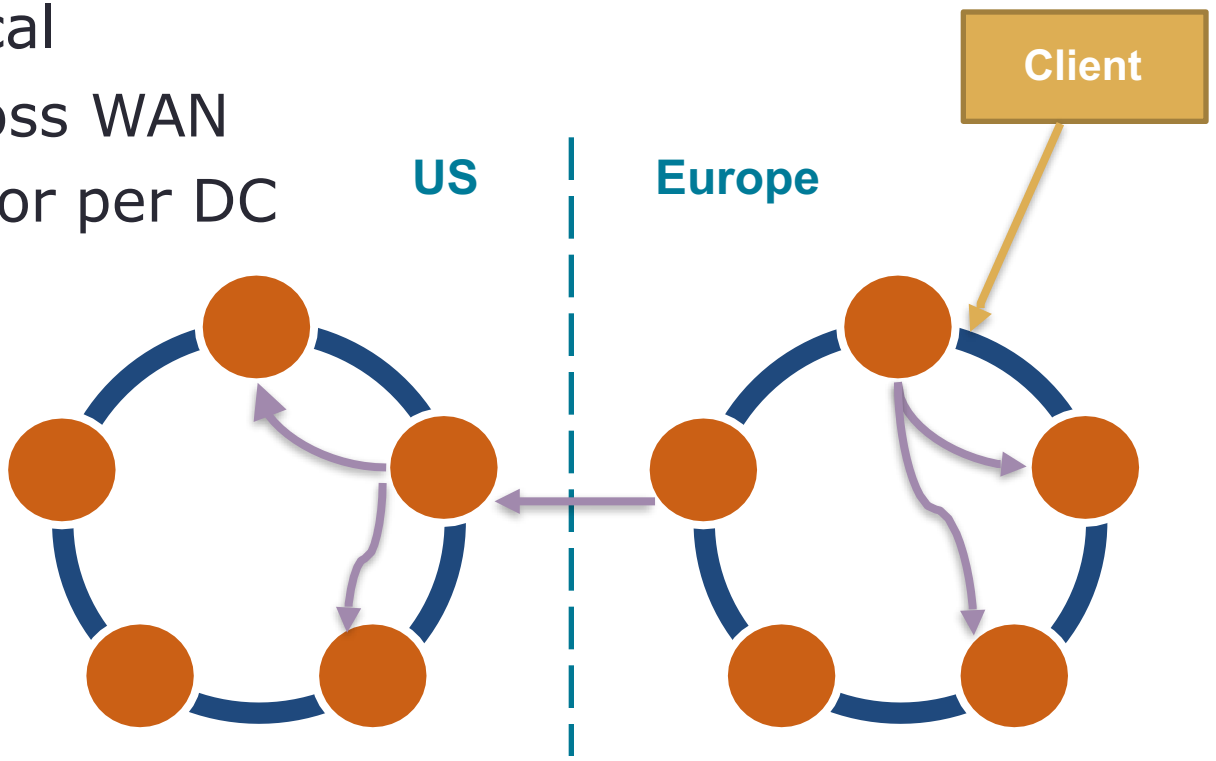
# Cassandra

- Fault Tolerant
  - Nodes Down != Database Down
  - Datacenter Down != Database Down



# Cassandra

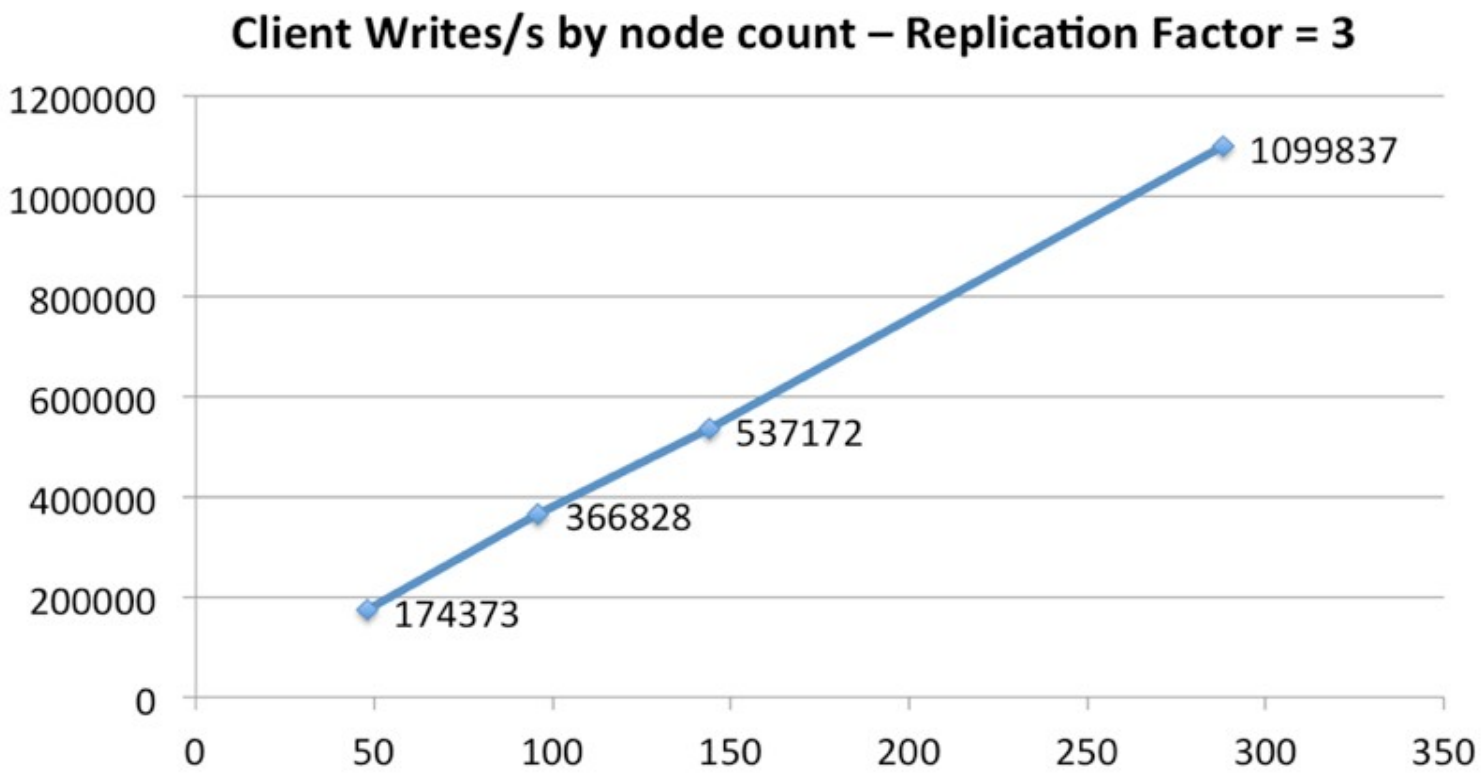
- Fully Replicated
- Clients write local
- Data syncs across WAN
- Replication Factor per DC



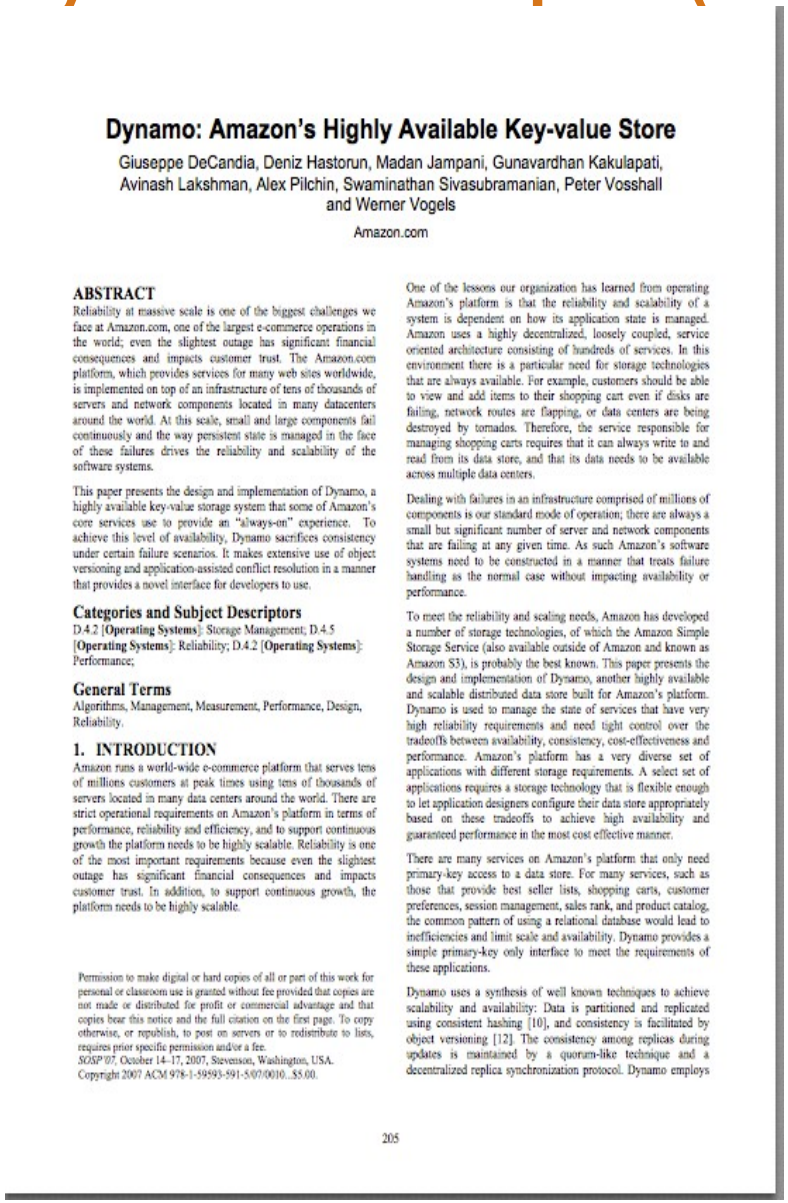
# Cassandra is great for...

- **Massive, linear scaling**
  (e.g. CERN hadron collider, Barracuda Networks)
- **Extremely heavy writes**
  (e.g. BlueMountain Capital – financial tick data)
- **High availability**
  (e.g. eBay, Eventbrite, Netflix, SoundCloud, HeathCare Anytime, Comcast, GoDaddy, Sony Entertainment Network)

# Scale-Up Linearity



# Dynamo Paper(2007)



**Dynamo: Amazon's Highly Available Key-value Store**

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels

Amazon.com

**ABSTRACT**

Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in the world; even the slightest outage has significant financial consequences and impacts customer trust. The Amazon.com platform, which provides services for many web sites worldwide, is implemented on top of an infrastructure of tens of thousands of servers and network components located in many datacenters around the world. At this scale, small and large components fail continuously and the way persistent state is managed in the face of these failures drives the reliability and scalability of the software systems.

This paper presents the design and implementation of Dynamo, a highly available key-value storage system that some of Amazon's core services use to provide an "always-on" experience. To achieve this level of availability, Dynamo sacrifices consistency under certain failure scenarios. It makes extensive use of object versioning and application-assisted conflict resolution in a manner that provides a novel interface for developers to use.

**Categories and Subject Descriptors**

D.4.2 [**Operating Systems**]: Storage Management; D.4.5 [**Operating Systems**]: Reliability; D.4.2 [**Operating Systems**]: Performance;

**General Terms**

Algorithms, Management, Measurement, Performance, Design, Reliability.

**1. INTRODUCTION**

Amazon runs a world-wide e-commerce platform that serves tens of millions customers at peak times using tens of thousands of servers located in many data centers around the world. There are strict operational requirements on Amazon's platform in terms of performance, reliability and efficiency, and to support continuous growth the platform needs to be highly scalable. Reliability is one of the most important requirements because even the slightest outage has significant financial consequences and impacts customer trust. In addition, to support continuous growth, the platform needs to be highly scalable.


*SOSP'07*, October 14–17, 2007, Stevenson, Washington, USA.


One of the lessons our organization has learned from operating Amazon's platform is that the reliability and scalability of a system is dependent on how its application state is managed. Amazon uses a highly decentralized, loosely coupled, service oriented architecture consisting of hundreds of services. In this environment there is a particular need for storage technologies that are always available. For example, customers should be able to view and add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being destroyed by tornados. Therefore, the service responsible for managing shopping carts requires that it can always write to and read from its data store, and that its data needs to be available across multiple data centers.

Dealing with failures in an infrastructure comprised of millions of components is our standard mode of operation; there are always a small but significant number of server and network components that are failing at any given time. As such Amazon's software systems need to be constructed in a manner that treats failure handling as the normal case without impacting availability or performance.

To meet the reliability and scaling needs, Amazon has developed a number of storage technologies, of which the Amazon Simple Storage Service (also available outside of Amazon and known as Amazon S3), is probably the best known. This paper presents the design and implementation of Dynamo, another highly available and scalable distributed data store built for Amazon's platform. Dynamo is used to manage the state of services that have very high reliability requirements and need tight control over the tradeoffs between availability, consistency, cost-effectiveness and performance. Amazon's platform has a very diverse set of applications with different storage requirements. A select set of applications requires a storage technology that is flexible enough to let application designers configure their data store appropriately based on these tradeoffs to achieve high availability and guaranteed performance in the most cost effective manner.

There are many services on Amazon's platform that only need primary-key access to a data store. For many services, such as those that provide best seller lists, shopping carts, customer preferences, session management, sales rank, and product catalog, the common pattern of using a relational database would lead to inefficiencies and limit scale and availability. Dynamo provides a simple primary-key only interface to meet the requirements of these applications.

Dynamo uses a synthesis of well known techniques to achieve scalability and availability: Data is partitioned and replicated using consistent hashing [10], and consistency is facilitated by object versioning [12]. The consistency among replicas during updates is maintained by a quorum-like technique and a decentralized replica synchronization protocol. Dynamo employs

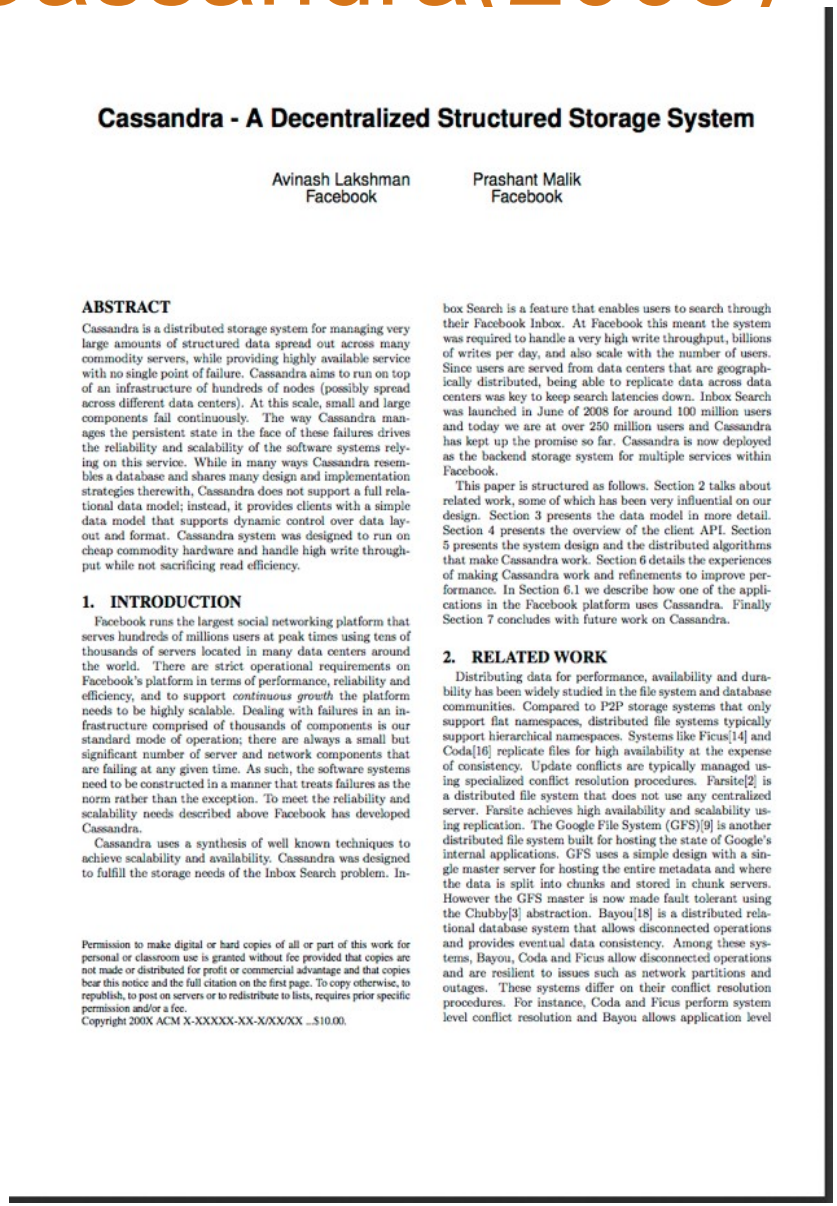
205

- How do we build a data store that is:
  - Reliable
  - Performant
  - "Always On"
- Nothing new and shiny
- 24 papers cited

amazon.com

Also the basis for Riak and Voldemort

# Cassandra(2008)



**Cassandra - A Decentralized Structured Storage System**

Avinash Lakshman
Facebook

Prashant Malik
Facebook

**ABSTRACT**

Cassandra is a distributed storage system for managing very large amounts of structured data spread out across many commodity servers, while providing highly available service with no single point of failure. Cassandra aims to run on top of an infrastructure of hundreds of nodes (possibly spread across different data centers). At this scale, small and large components fail continuously. The way Cassandra manages the persistent state in the face of these failures drives the reliability and scalability of the software systems relying on this service. While in many ways Cassandra resembles a database and shares many design and implementation strategies therewith, Cassandra does not support a full relational data model; instead, it provides clients with a simple data model that supports dynamic control over data layout and format. Cassandra system was designed to run on cheap commodity hardware and handle high write throughput while not sacrificing read efficiency.

**1. INTRODUCTION**

Facebook runs the largest social networking platform that serves hundreds of millions users at peak times using tens of thousands of servers located in many data centers around the world. There are strict operational requirements on Facebook's platform in terms of performance, reliability and efficiency, and to support *continuous growth* the platform needs to be highly scalable. Dealing with failures in an infrastructure comprised of thousands of components is our standard mode of operation; there are always a small but significant number of server and network components that are failing at any given time. As such, the software systems need to be constructed in a manner that treats failures as the norm rather than the exception. To meet the reliability and scalability needs described above Facebook has developed Cassandra.

Cassandra uses a synthesis of well known techniques to achieve scalability and availability. Cassandra was designed to fulfill the storage needs of the Inbox Search problem. Inbox Search is a feature that enables users to search through their Facebook Inbox. At Facebook this meant the system was required to handle a very high write throughput, billions of writes per day, and also scale with the number of users. Since users are served from data centers that are geographically distributed, being able to replicate data across data centers was key to keep search latencies down. Inbox Search was launched in June of 2008 for around 100 million users and today we are at over 250 million users and Cassandra has kept up the promise so far. Cassandra is now deployed as the backend storage system for multiple services within Facebook.

This paper is structured as follows. Section 2 talks about related work, some of which has been very influential on our design. Section 3 presents the data model in more detail. Section 4 presents the overview of the client API. Section 5 presents the system design and the distributed algorithms that make Cassandra work. Section 6 details the experiences of making Cassandra work and refinements to improve performance. In Section 6.1 we describe how one of the applications in the Facebook platform uses Cassandra. Finally Section 7 concludes with future work on Cassandra.

**2. RELATED WORK**

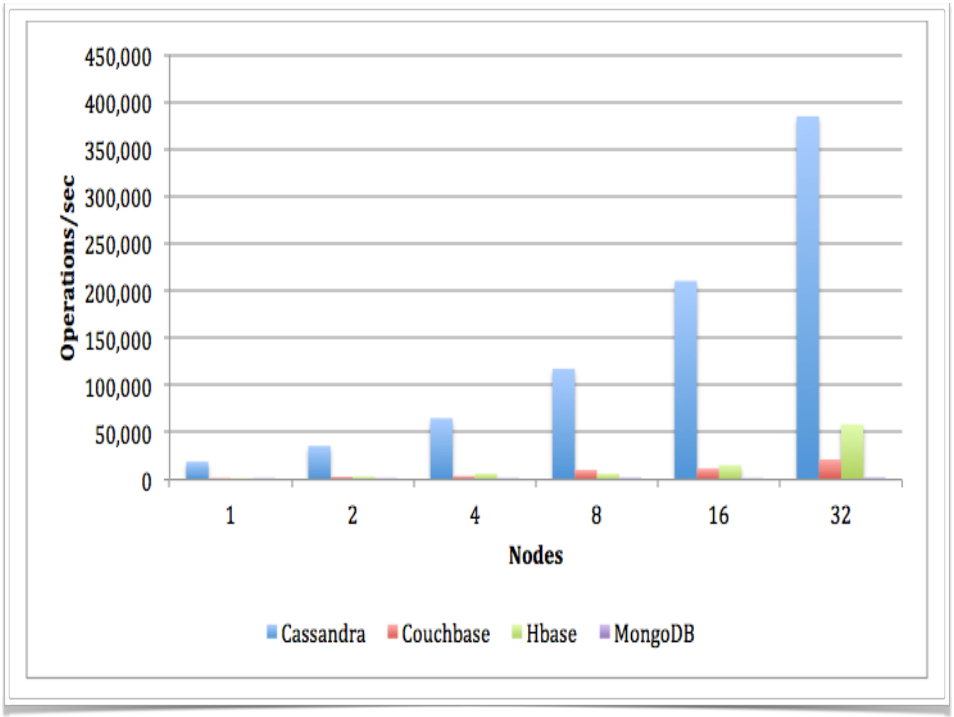
Distributing data for performance, availability and durability has been widely studied in the file system and database communities. Compared to P2P storage systems that only support flat namespaces, distributed file systems typically support hierarchical namespaces. Systems like Ficus[14] and Coda[16] replicate files for high availability at the expense of consistency. Update conflicts are typically managed using specialized conflict resolution procedures. Farsite[2] is a distributed file system that does not use any centralized server. Farsite achieves high availability and scalability using replication. The Google File System (GFS)[9] is another distributed file system built for hosting the state of Google's internal applications. GFS uses a simple design with a single master server for hosting the entire metadata and where the data is split into chunks and stored in chunk servers. However the GFS master is now made fault tolerant using the Chubby[3] abstraction. Bayou[18] is a distributed relational database system that allows disconnected operations and provides eventual data consistency. Among these systems, Bayou, Coda and Ficus allow disconnected operations and are resilient to issues such as network partitions and outages. These systems differ on their conflict resolution procedures. For instance, Coda and Ficus perform system level conflict resolution and Bayou allows application level



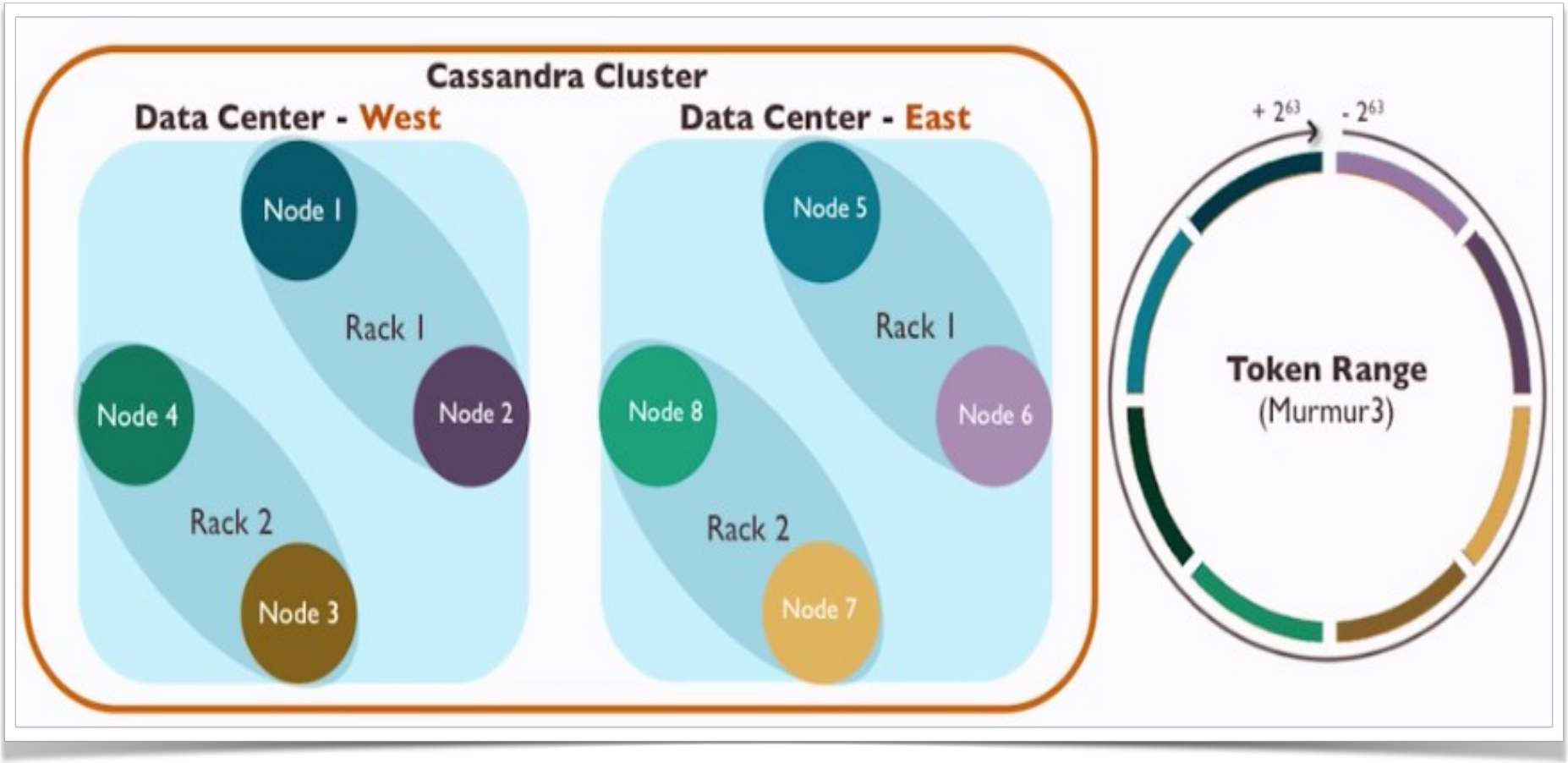
- Distributed features of Dynamo
- Data Model and storage from BigTable
- February 17, 2010 it graduated to a top-level Apache project

# CASSANDRA

- Fast Distributed NoSQL Database
- High Availability
- Linear Scalability => Predictability
- No SPOF
- Multi-DC
- Horizontally scalable
- Not a drop in replacement for RDBMS

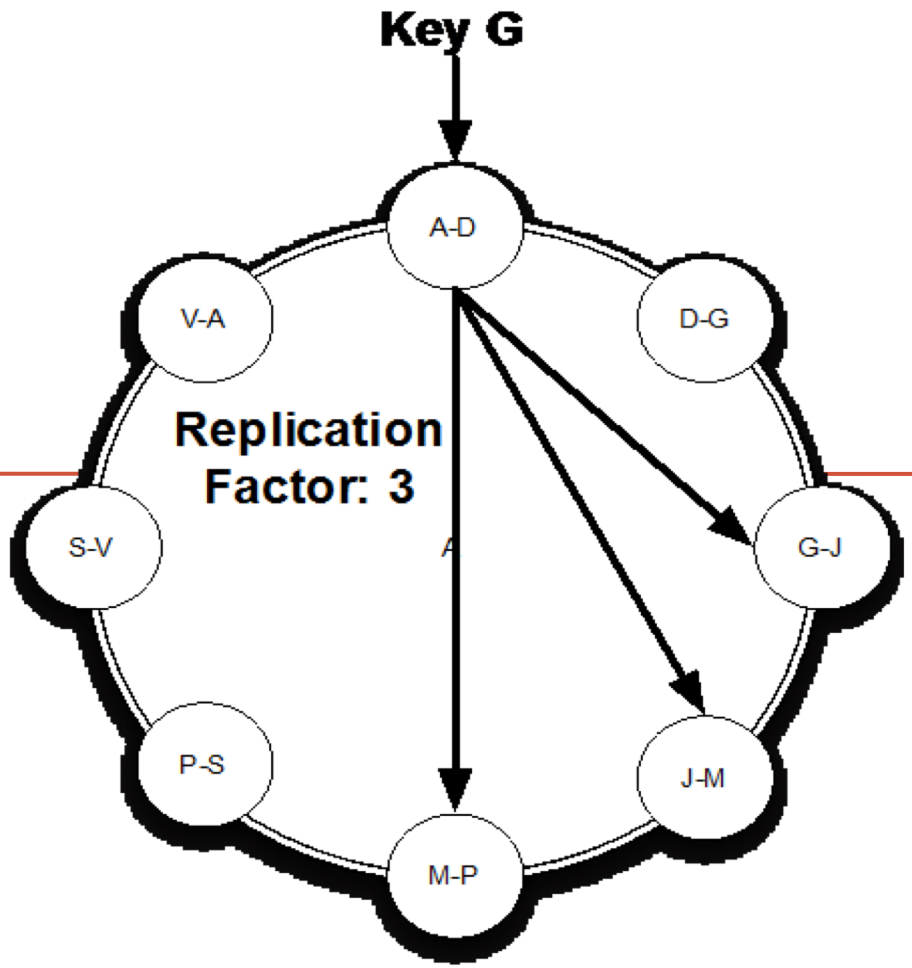


# CASSANDRA CLUSTER



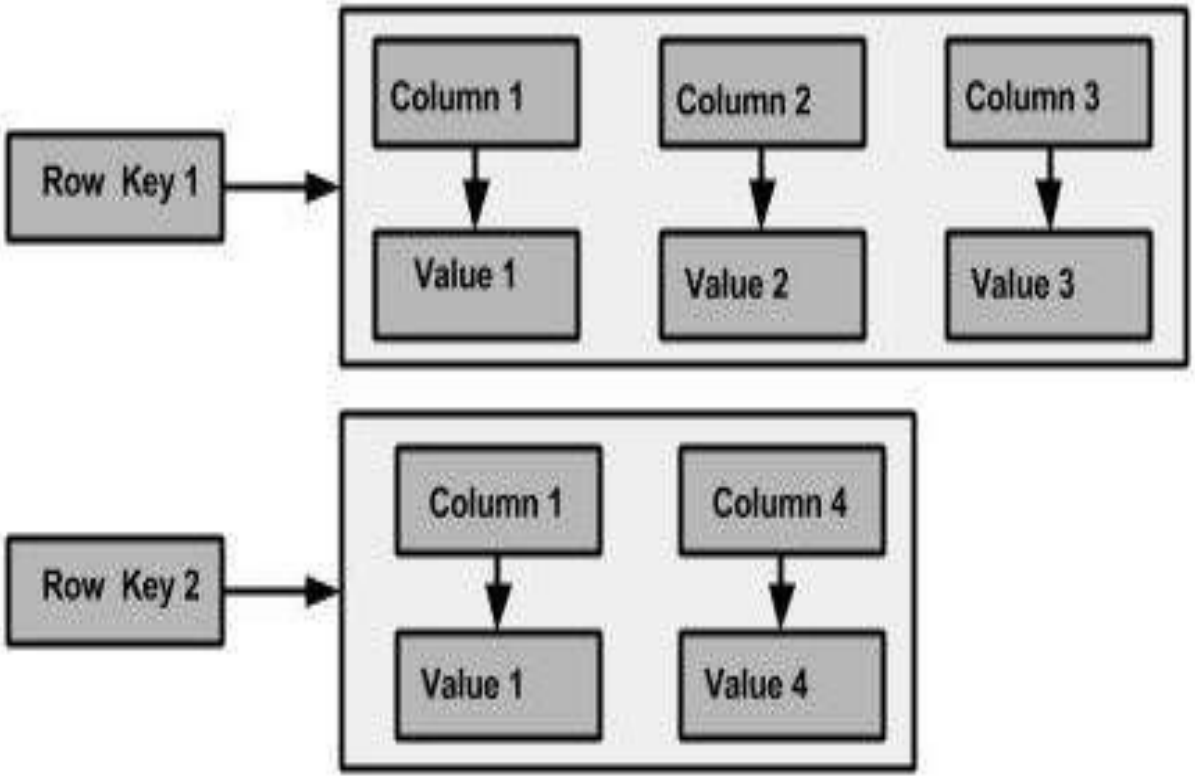
# REPLICATION FACTOR

How many copies (replicas) for your data



# CASSANDRA DATA MODEL

- Query driven data model
- Column family non relational db



# CQL

**Familiar row-column SQL-like approach.**

```
CREATE TABLE
  users (   id UUID,
  name VARCHAR,
  surname
  VARCHAR,
  birthdate
  TIMESTAMP,
  PRIMARY KEY(id)
);
```

```
INSERT INTO users (id, name, surname,
  birthdate)  VALUES (uuid(),'Carlos','Alonso',
  '1985-03-19');
```

```
SELECT * FROM users WHERE
id = 'f81d4fae-7dec-11d0-a765-
00a0c91e6bf6';
```

```
ALTER TABLE users ADD address
VARCHAR;
```

# CASSANDRA:YES

- If you need:
  - No SPOF
  - Linear horizontal scalability in commodity hardware
  - Real-time writes
  - Reliable data replication across distributed data centres
  - Clearly defined schema in a NoSQL environment

# CASSANDRA: NO

- If you need:
  - ACID transactions with rollback
  - Justification for high-end software

## What do consistency, availability and partition tolerance mean?

Consistency: All clients have the exact same value for the whole data set at any given point.

Availability: All clients can read and write to the system at any given point.

Partition tolerance: Whether or not the system tolerates a node being disconnected from the system.

## Where does Cassandra fit within the CAP Theorem?

AP

Cassandra trades off consistency in order to guarantee availability and partition tolerance, but in a configurable way, so it's up to the developer where to sit for each query.

## Which are the technological roots of Cassandra?

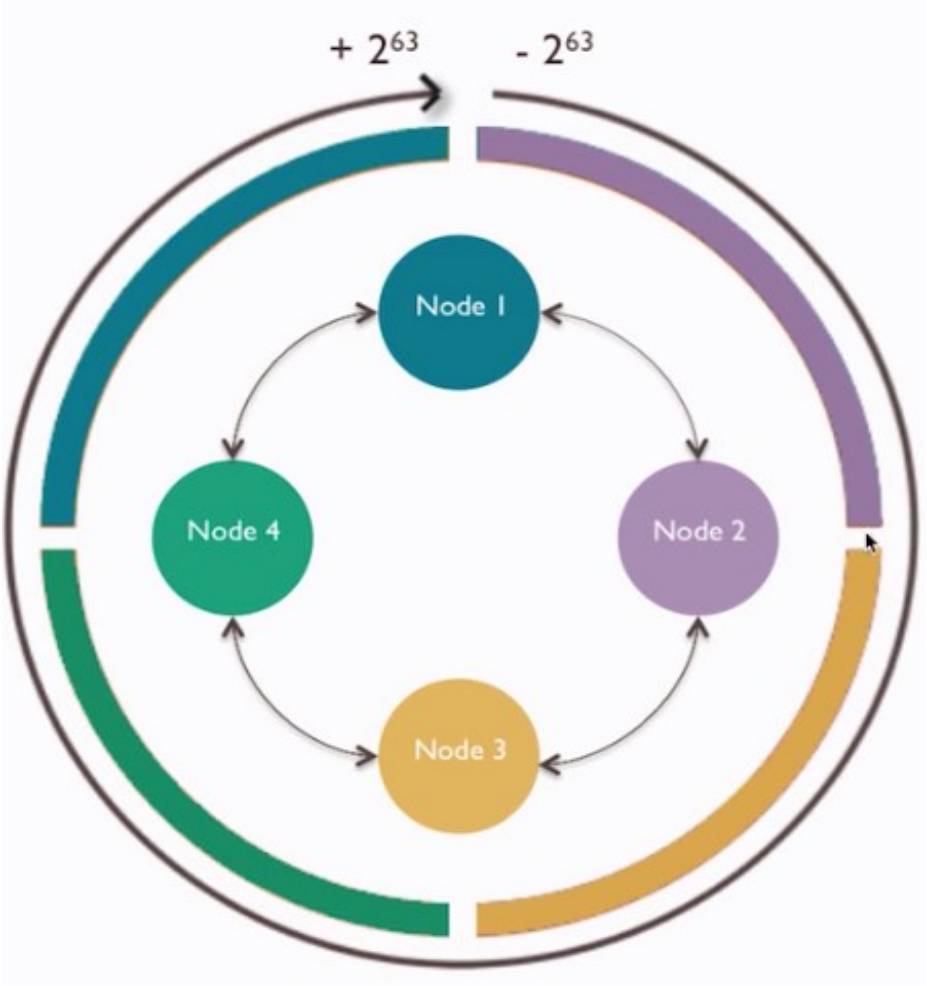
Google BigTable and Amazon Dynamo pulled together by developers at Facebook

**What technology does Cassandra use to model data?**

CQL: Cassandra Query Language

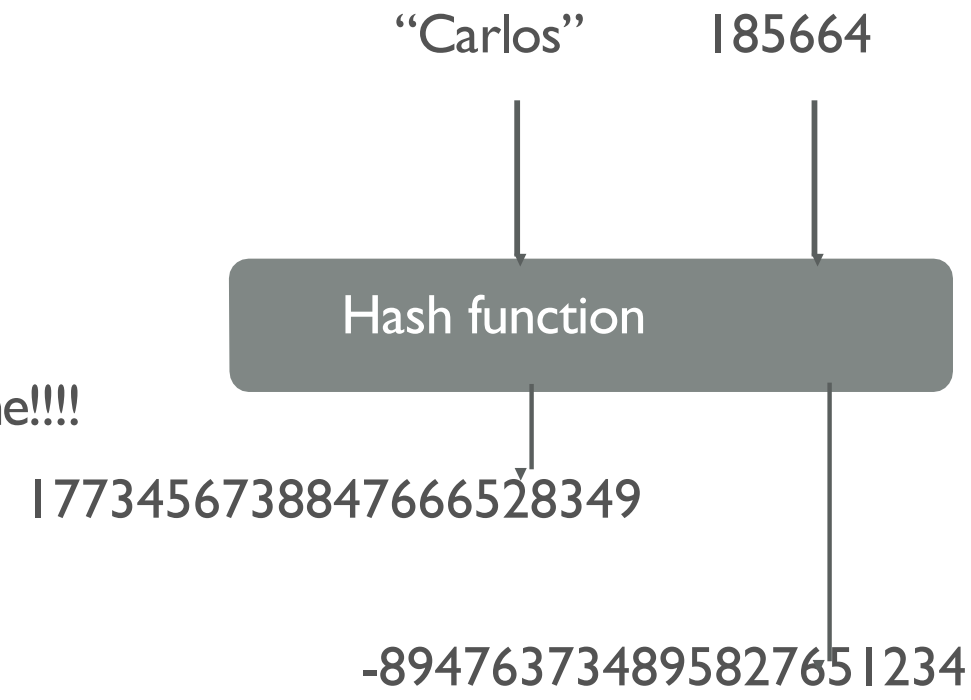
# CONSISTENT HASHING

- Data is stored in partitions, identified by a unique token within the range (-2^63 - 2^63)
- Nodes contain partition ranges.

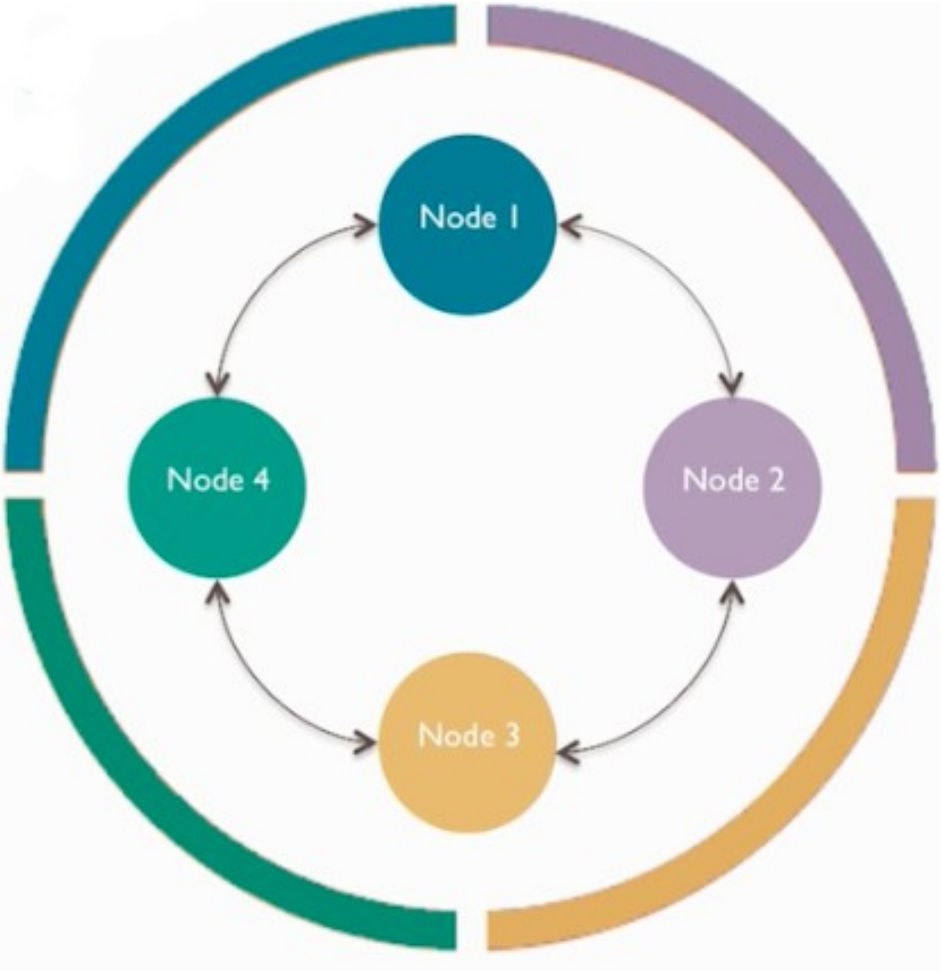


# THE PARTITIONER

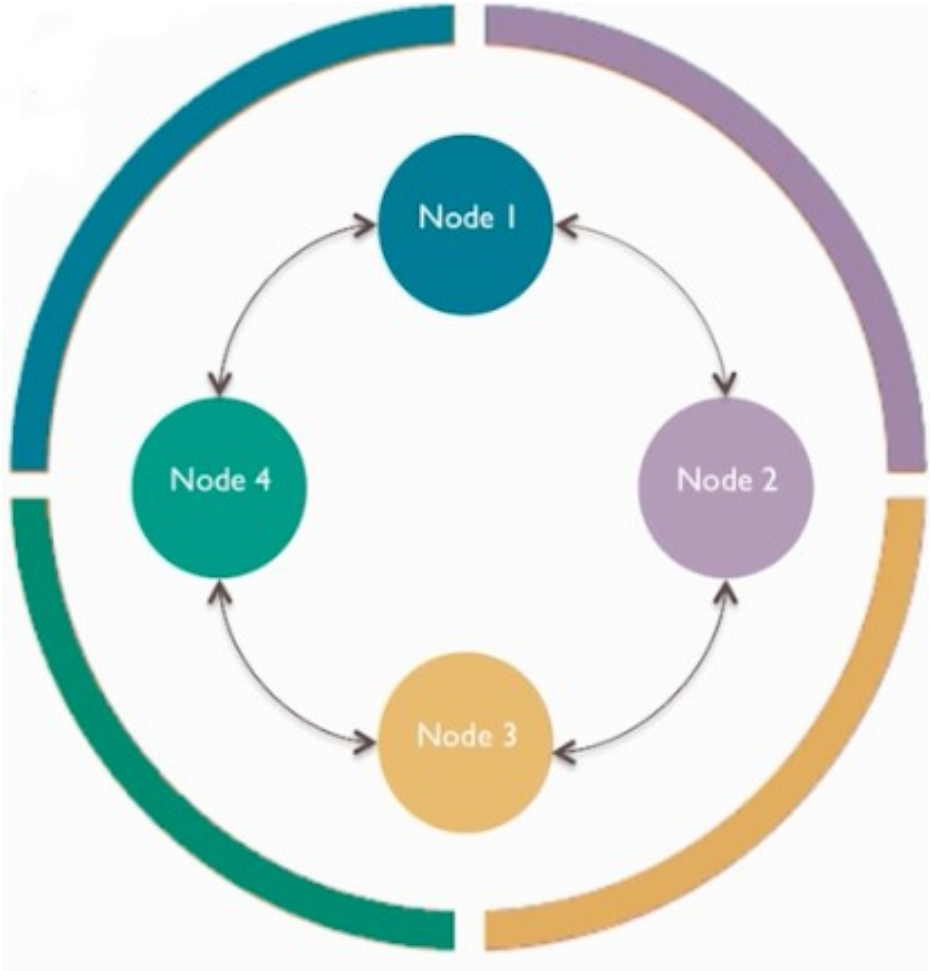
- System running on each node that computes hashes through a hash function.
- Various partitioners available.
  - Default is murmur3
- All nodes **MUST** use the same!!!!

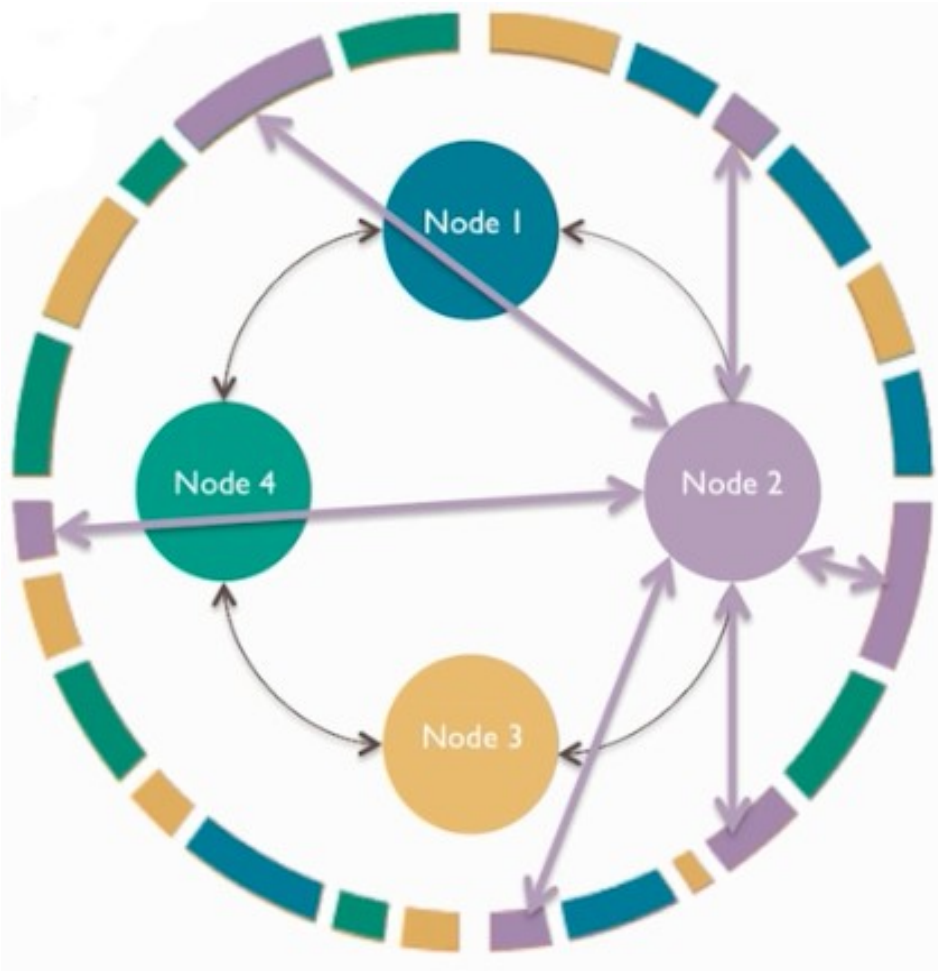


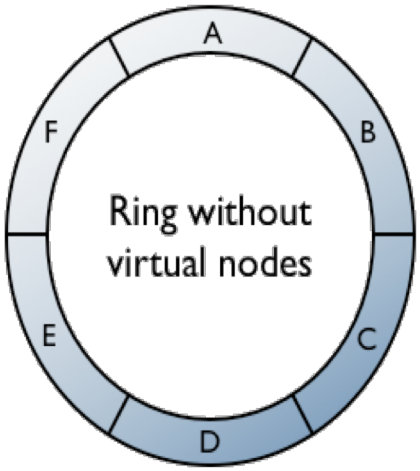
# VNODES

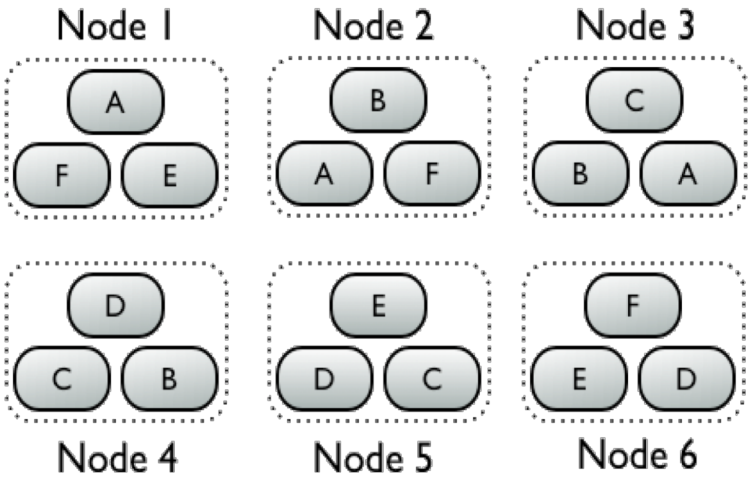


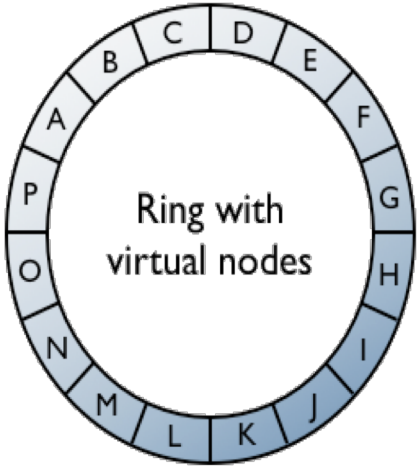
# VNODES

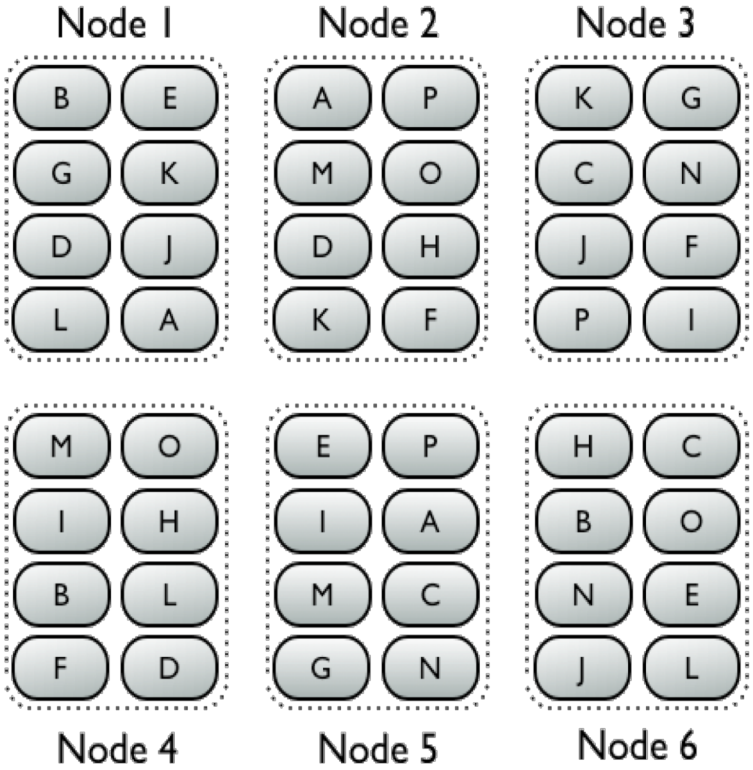



A
B
C
D
E
F
Ring without
virtual nodes


Node 1
A
F
E
Node 2
B
A
F
Node 3
C
B
A
D
C
B
Node 4
E
D
C
Node 5
F
E
D
Node 6


A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Ring with
virtual nodes


Node 1
B
E
G
K
D
J
L
A
Node 2
A
P
M
O
D
H
K
F
Node 3
K
G
C
N
J
F
P
I
M
O
I
H
B
L
F
D
Node 4
E
P
I
A
M
C
G
N
Node 5
H
C
B
O
N
E
J
L
Node 6

# Write Behavior

1. Client Writes to any Cassandra Node
2. Coordinator Node replicates to nodes and Zones
3. Nodes return ack to coordinator
4. Coordinator returns ack to client
5. Data written to internal commit log disk


If a node goes offline, hinted handoff completes the write when the node comes back up.

Requests can choose to wait for one node, a quorum, or all nodes to ack the write

SSTable disk writes and compactions occur asynchronously

# CQL
# Cassandra Query Language

# Terminology

- Keyspace
  - Keyspace is like RDBMS Database or Schema
- Table, Row, Column
  - Like RDBMS, Cassandra uses tables, having rows with columns to store data
- Partition (Key)
  - Partitions are grouping of table rows across multiple Cassandra nodes
- Clustering Key (Optional)
  - Clusters are groupings of table rows on a node ordered by some attribute(s)

# For Example

```
CREATE KEYSPACE packagetracker WITH REPLICATION = {
'class' :  'SimpleStrategy', 'replication_factor' : 1 };

CREATE KEYSPACE packagetracker WITH REPLICATION = {
'class' :  'NetworkTopologyStrategy', 'dc1' : 2, 'dc2' : 2};

CREATE TABLE events (
package_id text,
status_timestamp timestamp,
location text,
 notes text,
 PRIMARY KEY (package_id, status_timestamp)
);
```

# Constructs

# Basic Data Types

- blob
- int
- text
- long
- uuid
- etc

# More Data Modeling Constructs

- Collections
  - map, set, list
- Time to live (TTL)
- Counters
- Secondary Indexes

# Approaching Data Modeling

- Model your queries, not your data
  - Optimize your data model for reads
- Don’t be afraid to denormalize
- You will get it wrong, iterate

- An Example:  User Logins

# The Table and a Query

What are the last 10 locations nickmbailey logged in from?

```
CREATE TABLE logins (
   user text,
   time timestamp,
   location text,
  PRIMARY KEY (user,   time));

SELECT time, location FROM logins
WHERE user = 'nickmbailey'
ORDER BY time DESC LIMIT 10;
```

# Partition Keys(s)

What are the last 10 locations nickmbailey logged in from?

```
CREATE TABLE logins (
   user text,
   time timestamp,
   location text,
  PRIMARY KEY (user,  time));

SELECT time, location FROM logins
WHERE user = 'nickmbailey'
ORDER BY time DESC LIMIT 10;
```

Partition Key

# Cluster Key(s)

What are the last 10 locations nickmbailey logged in from?

```
CREATE TABLE logins (
   user text,
   time timestamp,
   location text,
  PRIMARY KEY (user, time));

SELECT time, location FROM logins
WHERE user = 'nickmbailey'
ORDER BY time DESC LIMIT 10;
```

Cluster Key

# Non-Key Attributes (Columns)

What are the last 10 locations nickmbailey logged in from?

```
CREATE TABLE logins (
   user text,
   time timestamp,
   location text,
  PRIMARY KEY (user,   time));

SELECT time, location FROM logins
WHERE user = 'nickmbailey'
ORDER BY time DESC LIMIT 10;
```

Other Columns

# Partition and Cluster Keys

- What are the last 10 locations nickmbailey logged in from?

- SELECT time, location FROM logins WHERE user = 'nickmbailey' ORDER BY time DESC LIMIT 10;

- CREATE COLUMN FAMILY logins (
  - user text, time timestamp,
  - location text,
  - PRIMARY KEY (user, time));

Partition Key

Cluster Key

| User | Time | Location |
|---|---|---|
| nickmbailey | 2013-07-19 09:22:18 | Austin, Texas |
| nickmbailey | 2013-07-19 14:49:27 | Blacksburg, Virginia |
| jsmith | 2013-07-20 07:59:34 | Atlanta, Georgia |

# Time-series data

- By far, the most common data model
- Event logs
- Metrics
- Sensor Data
- Etc

# Another Query

When was the last time nickmbailey logged in from San Francisco, California?

```
SELECT time FROM logins WHERE user = 'nickmbailey'
and location='San Francisco, California';
```

| User | Time | Location |
|---|---|---|
| nickmbailey | 2013-07-19 09:22:18 | Austin, Texas |
| nickmbailey | 2013-07-19 14:49:27 | Blacksburg, Virginia |
| nickmbailey | 2013-07-19 14:49:27 | Austin, Texas |
| nickmbailey | 2013-05-19 14:49:27 | Austin, Texas |
| nickmbailey | 2013-04-19 14:49:27 | San Francisco, California |
| ... | ... | ... |
| jsmith | 2013-07-20 07:59:34 | Atlanta, Georgia |

# Another Query

When was the last time nickmbailey logged in from Austin, Texas?

```
SELECT time FROM logins_by_location WHERE user =
'nickmbailey'  and location='San Francisco, California';

CREATE COLUMN FAMILY logins_by_location
  (  user text,
  time timestamp,
  location text,
  PRIMARY KEY (user, location));
```

# Another Query

When was the last time nickmbailey logged in from Austin, Texas?

```
SELECT time FROM logins_by_location WHERE user =
'nickmbailey'  and location='San Francisco, California';

CREATE COLUMN FAMILY logins_by_location
  (  user text,
  time timestamp,
  location text,
  PRIMARY KEY (user, location));
```

| User | Location | Time |
|---|---|---|
| nickmbailey | Austin, Texas | 2013-07-19 09:22:18 |
| nickmbailey | Blacksburg, Virginia | 2013-07-19 14:49:27 |
| nickmbailey | San Francisco, California | 2013-07-19 14:49:27 |

# Denormalize

- Create materialized views of the same data to support different queries
- Storage space is cheap, Cassandra is fast

# Schema Definition (DDL)

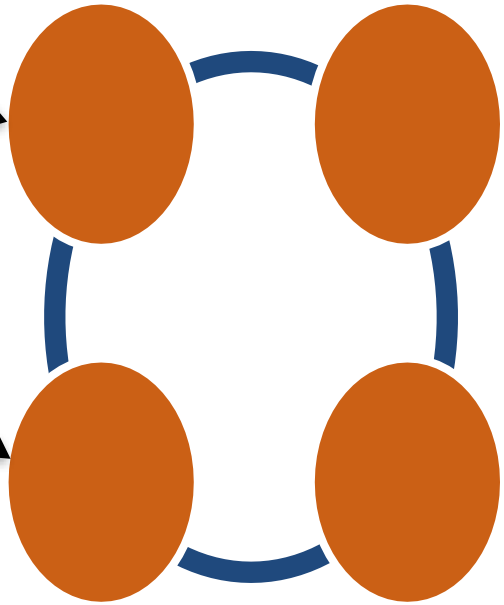
- Easy to define tables for storing data
- First part of Primary Key is the **Partition Key**

```
CREATE TABLE videos (
    videoid uuid,
    userid uuid,
    name text,
    description text,
    tags set<text>,
    added_date timestamp,
    PRIMARY KEY (videoid)
);
```

# Schema Definition (DDL)

```
CREATE TABLE videos (
    videoid uuid,
    userid uuid,
    name text,
    description text,
    tags set<text>,
    added_date timestamp,
    PRIMARY KEY (videoid)
);
```

| videoid | name | ... |
|---|---|---|
| 689d56e5-... | Keyboard Cat | ... |
| 93357d73-... | Nyan Cat | ... |
| d978b136-... | Original Grumpy Cat | ... |



# Clustering Columns

- Second part of Primary Key is **Clustering Columns**

```
CREATE TABLE comments_by_video (
    videoid uuid,
    commentid timeuuid,
    userid uuid,
    comment text,
    PRIMARY KEY (videoid, commentid)
) WITH CLUSTERING ORDER BY (commentid DESC);
```

- Clustering columns affect ordering of data (on disk)
- Multiple rows per partition

# Clustering Columns

DATASTAX

| videoid | commentid | ... |
|---|---|---|
| 689d56e5-... | 8982d56e5... | ... |
| 689d56e5-... | 93822df62... | ... |
| 689d56e5-... | 22dt62f69... | ... |
| 93357d73-... | 8319af913... | |

# Inserts and Updates

- Use INSERT or UPDATE to add and modify data

```
INSERT INTO comments_by_video (
    videoid, commentid, userid, comment)
VALUES (
    '0fe6a...', '82be1...', 'ac346...', 'Awesome!');
```

```
UPDATE comments_by_video
SET userid = 'ac346...', comment = 'Awesome!'
WHERE videoid = '0fe6a...' AND commentid =
'82be1...';
```

# Deletes

- Can specify a Time to Live (TTL) in seconds when doing an INSERT or UPDATE

```
INSERT INTO comments_by_video
( ... )
VALUES ( ... )
USING TTL 86400;
```

- Use DELETE statement to remove data

```
DELETE FROM comments_by_video
WHERE videoid = '0fe6a...' AND commentid =
'82be1...';
```

# Querying

- Use SELECT to get data from your tables

```
SELECT * FROM comments_by_video
WHERE videoid = 'a67cd...'
LIMIT 10;
```

- Always include **Partition Key** and optionally **Clustering Columns**
- Can use ORDER BY and LIMIT

- Use range queries (for example, by date) to slice partitions

# Cassandra Data Modeling

- Hmmm, looks like SQL, I know that…

# Cassandra Data Modeling

- Requires a different mindset than RDBMS modeling
- Know your data and your queries up front
- Queries drive a lot of the modeling decisions (i.e. “table per query” pattern)
- Denormalize/Duplicate data at write time to do as few queries as possible come read time
- Remember, disk is cheap and writes in Cassandra are **FAST**
-

# Other Data Modeling Concepts

- Lightweight Transactions
- JSON
- User Defined Types
- User Defined Functions

# Accessing CQL

- Common ways to access CQL are:
  - For open source and commercial installations, start cqlsh, the Python-based command-line client, on the command line of a Cassandra node
  - For commercial installations only, use DataStax DevCenter, a graphical user interface

# CQLSH

Our data management and first exploration tool

# CQLSH

- **help:** shows available cqlsh + CQL commands
- **DESCRIBE:** shows information of the arguments
- **SOURCE:** executes a file containing CQL statements
- **TRACING:** enables/disables the tracing mode
- **SELECT, ALTER, INSERT, …**

# CQL Lexical Structure

- CQL input consists of statements
- Like SQL, statements change data, look up data, store data, or change the way data is stored
- Statements end in a semicolon (;)
- For example, the following is valid CQL syntax:

```
SELECT * FROM MyTable;

UPDATE MyTable
  SET SomeColumn = 'SomeValue'
  WHERE columnName = ‘SomeColumnName’;
```

- This is a sequence of two CQL statements

# CQL Lexical Structure

- Identifiers created using CQL are case-insensitive unless enclosed in double quotation marks
- If you enter names for these objects using any uppercase letters, Cassandra stores the names in lowercase
- You can force the case by using double quotation marks. For example:

```
CREATE TABLE test (
  Foo int PRIMARY KEY,        <= Foo is stored as foo
  "Bar" int                   <= Bar is stored as Bar
);
```

# CQL Lexical Structure

- What Works and What Doesn't

| Queries that Work | Queries that Don't Work |
|---|---|
| SELECT foo FROM . . . | SELECT "Foo" FROM . . . |
| SELECT Foo FROM . . . | SELECT "BAR" FROM . . . |
| SELECT FOO FROM . . . | SELECT bar FROM . . . |
| SELECT "Bar" FROM . . . | SELECT Bar FROM . . . |
| SELECT "foo" FROM . . . | SELECT "bar" FROM . . . |

- SELECT "foo" FROM ... works because internally, Cassandra stores foo in lowercase

# CQL Lexical Structure

- CQL keywords are case-insensitive
- For example, the keywords SELECT and select are equivalent

# CQL Code Comments

- Use the following notation to include comments in CQL code
- For a single line or end of line put a double hyphen before the text, this comments out the rest of the line:

**select** * **from** cycling.route; -- *End of line comment*

- For a single line or end of line put a double forward slash before the text, this comments out the rest of the line:

**select** * **from** cycling.route; // **End of** line comment

- For a block of comments put a forward slash asterisk at the beginning of the comment and then asterisk forward slash at the end

- */* This is the first line of a comment that spans multiple lines */*
- **select** * **from** cycling.route;

# CREATE KEYSPACE

- Creates a top-level namespace
- Configure the replica placement strategy, replication factor, and durable writes setting

```
CREATE  KEYSPACE [IF NOT EXISTS] keyspace_name
   WITH REPLICATION = {
      'class' : 'SimpleStrategy', 'replication_factor' : N }
     | 'class' : 'NetworkTopologyStrategy',
       'dc1_name' : N [, ...]
   }
   [AND DURABLE_WRITES =  true|false] ;
```

# CREATE KEYSPACE

- CREATE KEYSPACE [IF NOT EXISTS]
- *keyspace_name*Keyspace names can have up to 48 alpha-numeric characters and contain underscores; only letters and numbers are supported as the first character
- Cassandra forces keyspace names to lowercase when entered without quotes
- If a keyspace with the same name already exists, an error occurs and the operation fails
  - Use IF NOT EXISTS to suppress the error message.

# CREATE KEYSPACE

- REPLICATION = { *replication_map* }
- The replication map determines how many copies of the data are kept in a given data center
- This setting impacts consistency, availability and request speed

Replication strategy class and factor settings

| Class | Replication factor | Value Description |
|---|---|---|
| 'SimpleStrategy' | 'replication_factor' : *N* | Assign the same replication factor to the entire cluster |
| 'NetworkTopologyStrategy' | '*datacenter_name*' : *N* | Assign replication factors to each data center in a comma separated list. Use in production environments and multi-DC test and development environments. |

# CREATE KEYSPACE

- Simple Topolgy syntax:

'class' : 'SimpleStrategy', 'replication_factor' : *N*

- Network Topology syntax

'class' : 'NetworkTopologyStrategy', '*dc1_name*' : *N* [, ...]

- DURABLE_WRITES = true|false
- Optionally bypass the commit log when writing to the keyspace by disabling durable writes.
- Default value is true.

# CREATE KEYSPACE

- Create cycling keyspace on a single node evaluation cluster:

**CREATE KEYSPACE** cycling **WITH REPLICATION** = { 'class' : 'SimpleStrategy', 'replication_factor' : 1 };

# CREATE KEYSPACE

- Create the cycling keyspace in an environment with mutliple data centers
- Set the replication factor for the Boston, Seattle, and Tokyo data centers

```
CREATE KEYSPACE "Cycling"
  WITH REPLICATION = {
   'class' : 'NetworkTopologyStrategy',
   'boston'  : 3 , // Datacenter 1
   'seattle' : 2 , // Datacenter 2
   'tokyo'   : 2 , // Datacenter 3
  };
```

# REPLICATION

How many copies of your data?

# WHY REPLICATION?

- Disaster recovery
- Bring data closer to users (to reduce latencies)
- Workload segregation (analytical vs transactional)

# REPLICATION

## Defined at keyspace level

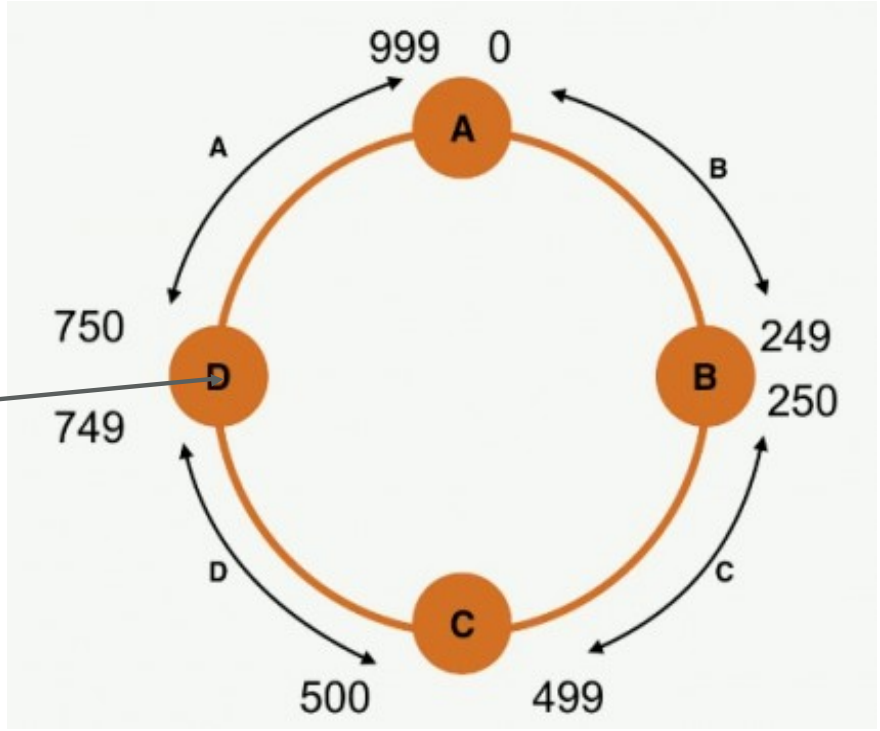
```
CREATE KEYSPACE <my_keyspace> WITH REPLICATION =
{"class":"SimpleStrategy","replication_factor":2 };
```

```
CREATE KEYSPACE <my_keyspace> WITH REPLICATION =
{"class":"NetworkTopologyStrategy",
"dc-east":2,"dc-west":3 };
```

# SIMPLESTRATEGY

CREATE KEYSPACE <my_keyspace> WITH REPLICATION = {"class":"**SimpleStrategy**","replication_factor":**3** };

Token:834

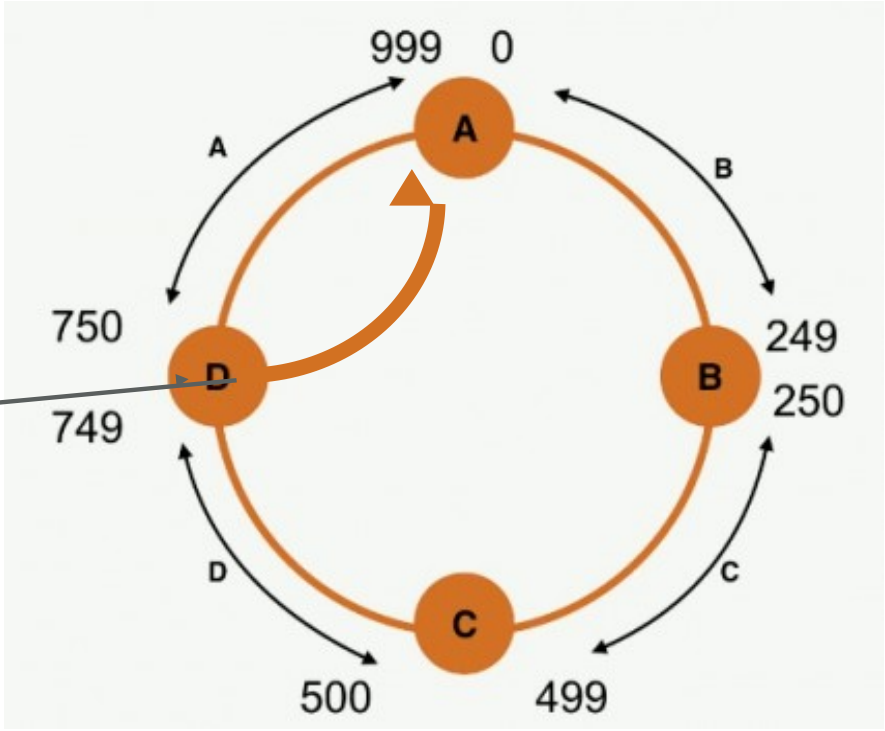



Client Driver

# SIMPLESTRATEGY

CREATE KEYSPACE <my_keyspace> WITH REPLICATION = {"class":"**SimpleStrategy**","replication_factor":**3** };

Token:834

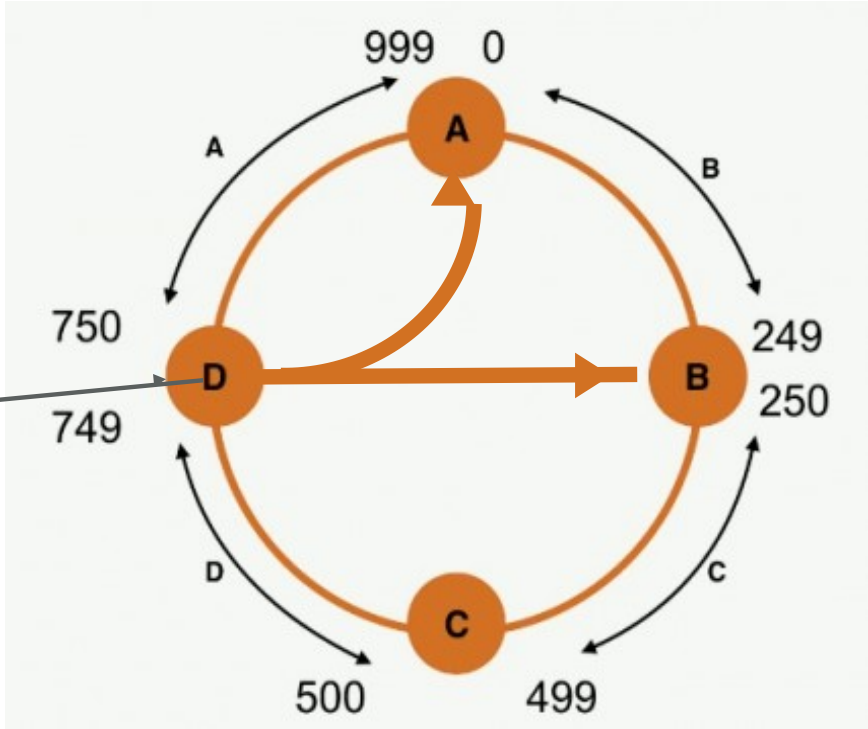



Client Driver

# SIMPLESTRATEGY

CREATE KEYSPACE <my_keyspace> WITH REPLICATION = {"class":"**SimpleStrategy**","replication_factor":**3** };

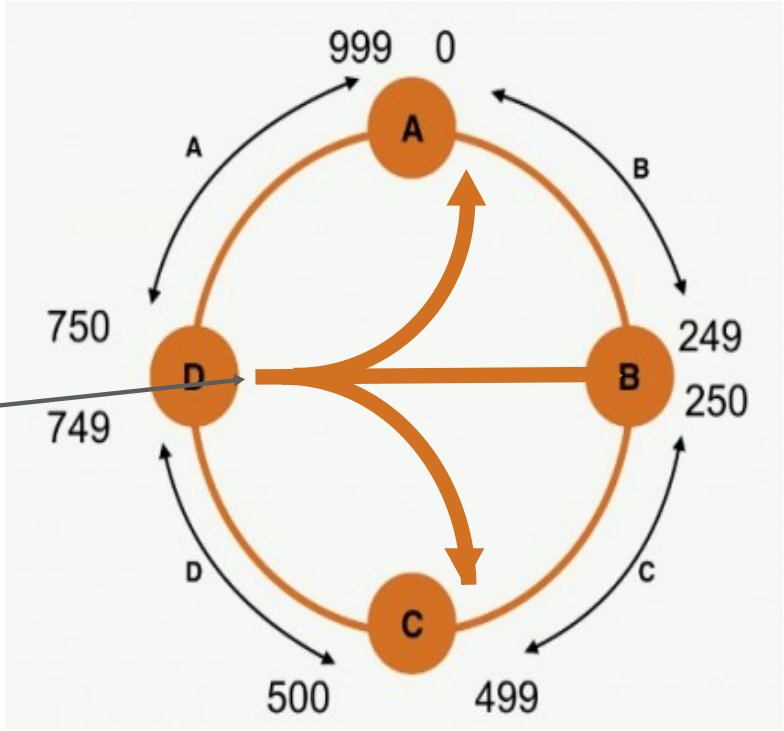
Token:834

# SIMPLESTRATEGY

CREATE KEYSPACE <my_keyspace> WITH REPLICATION = {"class":"**SimpleStrategy**","replication_factor":**3** };

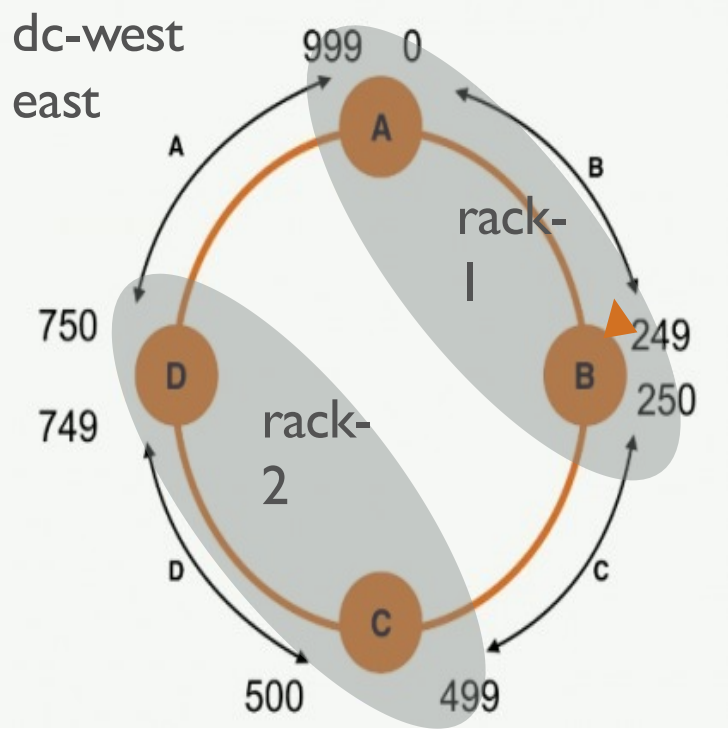
Token:834

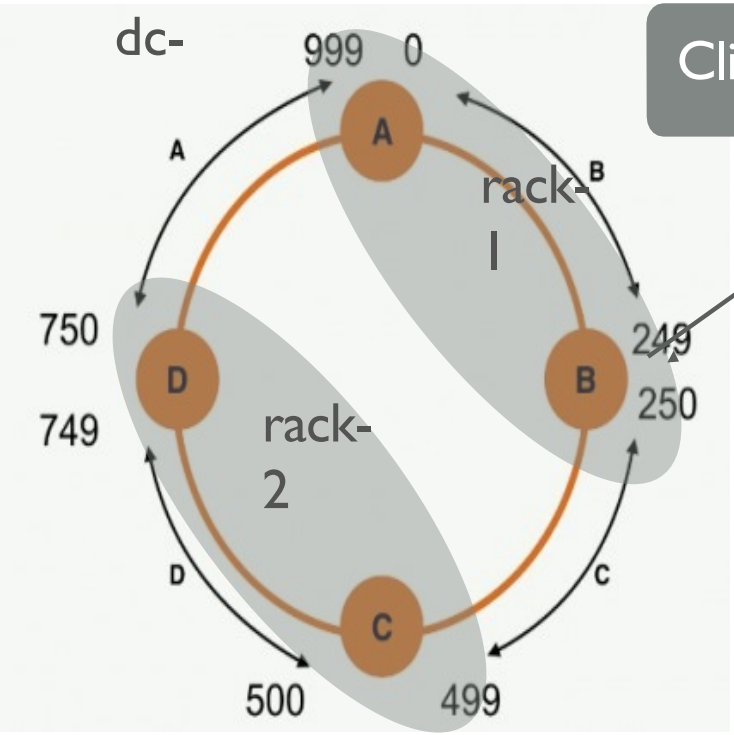



Client Driver

# NETWORKTOPOLOGYSTRATEGY

CREATE KEYSPACE <my_keyspace> WITH REPLICATION = {"class":"**NetworkTopologyStrategy**", "dc-east": **2**,"dc-west": **3** };



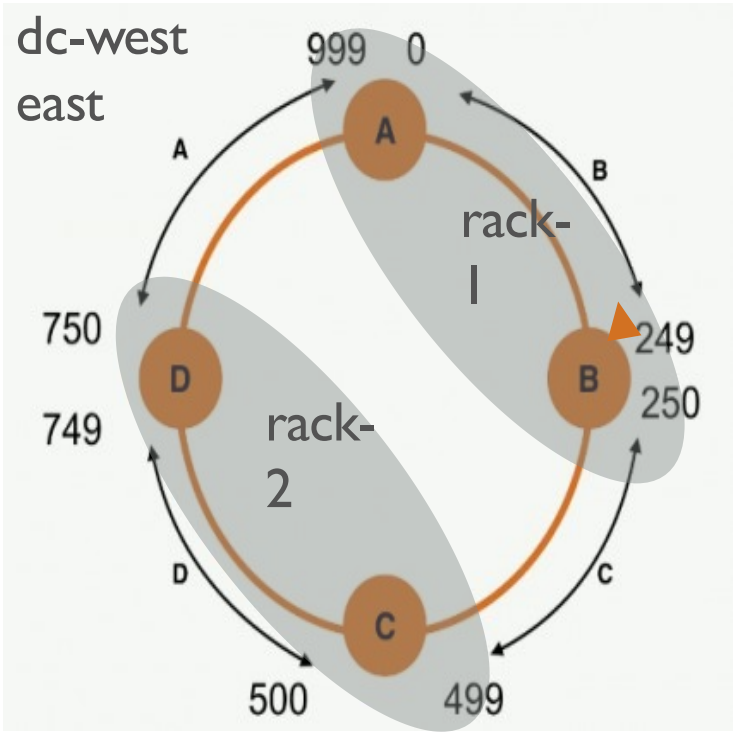

Client Driver

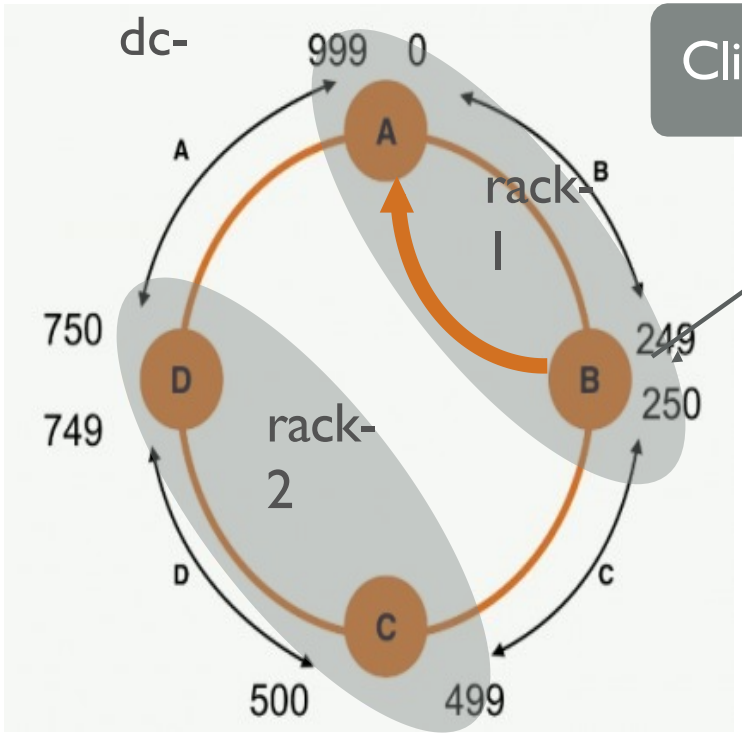
Token:834

@C
alon
so

# NETWORKTOPOLOGYSTRATEGY

```
CREATE KEYSPACE <my_keyspace> WITH REPLICATION =
{"class":"NetworkTopologyStrategy", "dc-
east": 2,"dc-west": 3 };
```



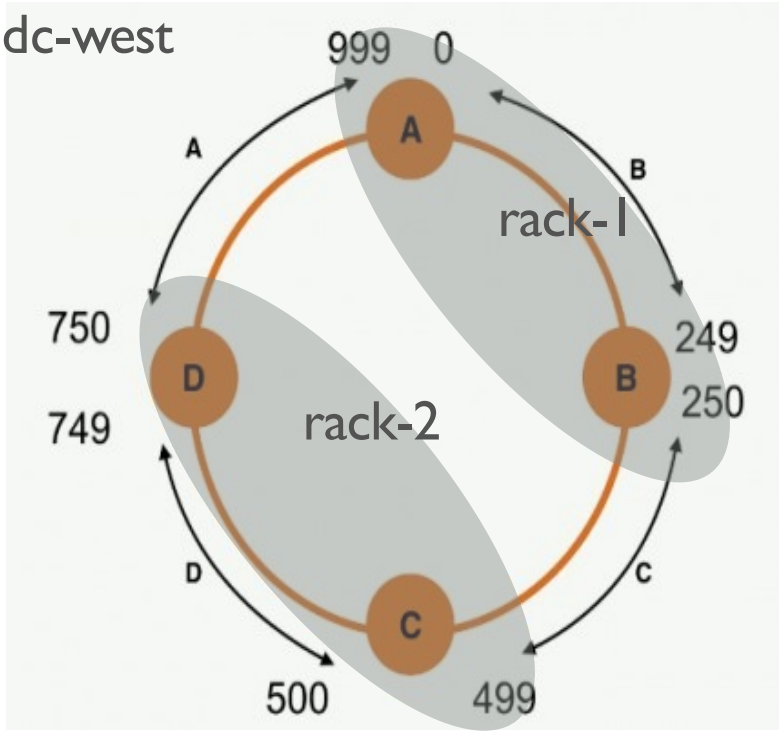



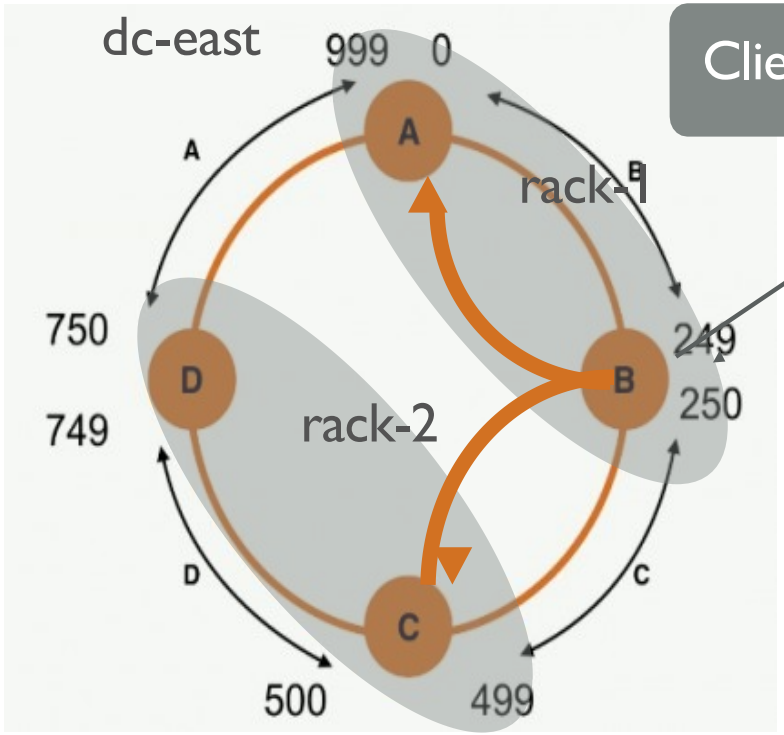

Client Driver

Token:834

# NETWORKTOPOLOGYSTRATEGY

CREATE KEYSPACE <my_keyspace> WITH REPLICATION =
{"class":"**NetworkTopologyStrategy**",
"dc-east": **2**,"dc-west": **3** };



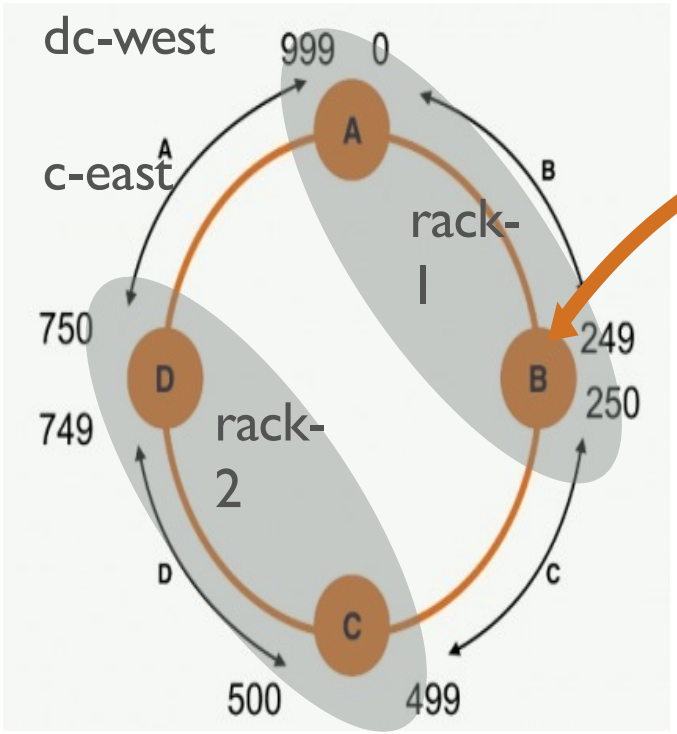



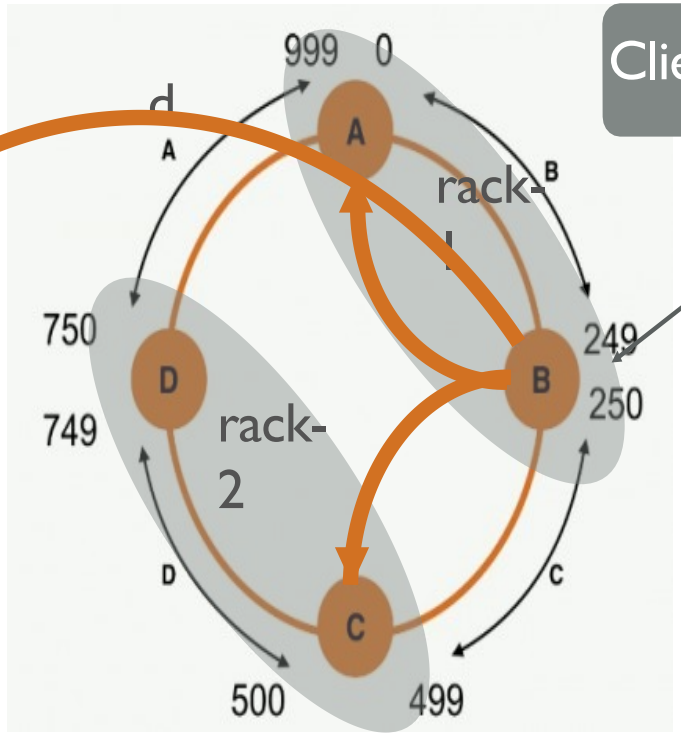

Client Driver

Token:834

# NETWORKTOPOLOGYSTRATEGY

CREATE KEYSPACE <my_keyspace> WITH REPLICATION =
{"class":"**NetworkTopologyStrategy**",
"dc-east": **2**,"dc-west": **3** };



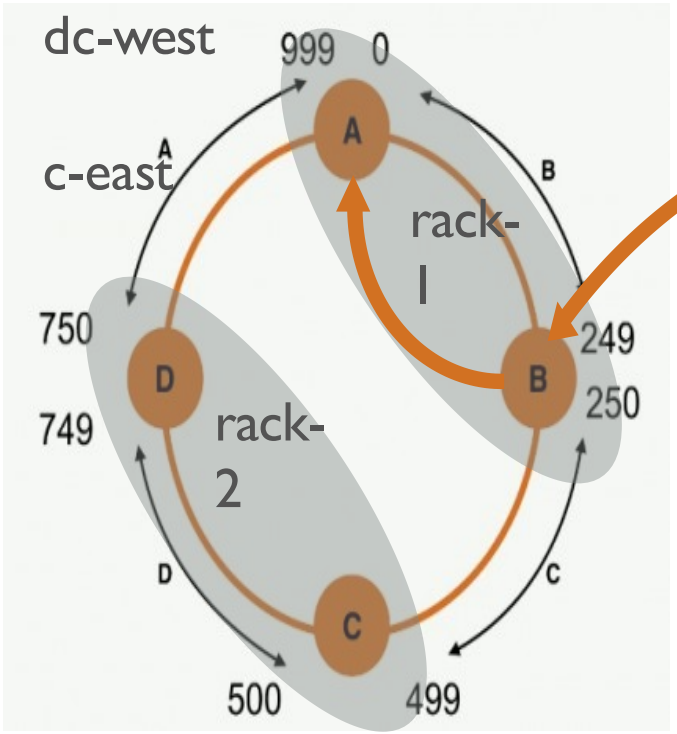



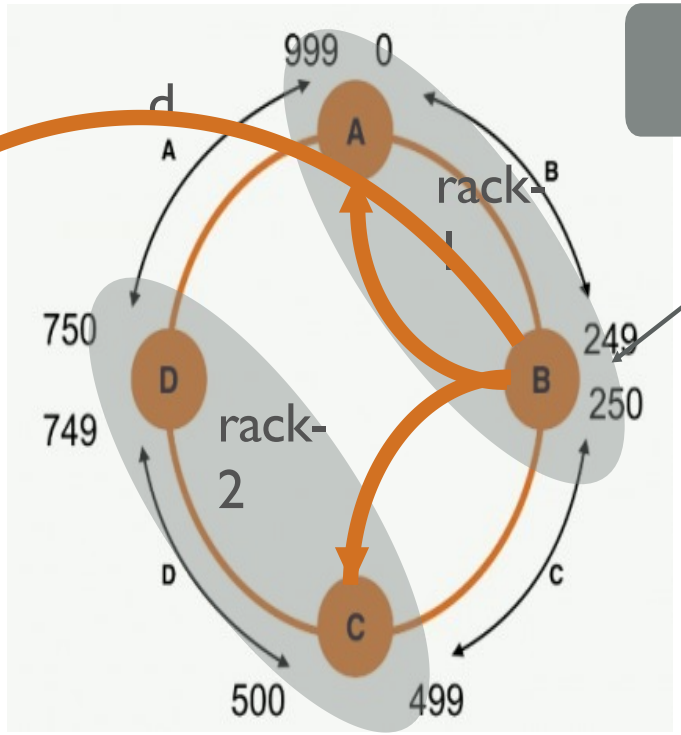

Client Driver

Token:
834

# NETWORKTOPOLOGYSTRATEGY

CREATE KEYSPACE <my_keyspace> WITH REPLICATION =
{"class":"**NetworkTopologyStrategy**",
"dc-east": **2**,"dc-west": **3** };




Client Driver

Token:
834

# CREATE TABLE

- The CREATE TABLE command creates a new table under a keyspace

```
CREATE TABLE [IF NOT EXISTS]
[keyspace_name.]table_name (
   column_definition [, ...]
   PRIMARY KEY (column_name [, column_name ...])
[WITH table_options
   | CLUSTERING ORDER BY (clustering_column_name
order])
   | ID = 'table_hash_tag'
   | COMPACT STORAGE]
```

# Primary Key | Single Key

- When the PRIMARY KEY is one column, append PRIMARY KEY to the end of the column definition
- This is only schema information required to create a table
- When there is one primary key, it is the partition key; the data is divided and stored by the unique values in this column:

*column_name cql_type_definition* PRIMARY KEY

- Alternatively, you can declare the primary key consisting of only one column in the same way as you declare a compound primary key

- Create the cyclist_name table with UUID as the primary key

```
CREATE TABLE cycling.cyclist_name (
   id UUID PRIMARY KEY,
   lastname text,
   firstname text
);
```

# Primary Key | Compound Key

- A compound primary key consists of more than one column
- The first column is the partition key, and the additional columns are clustering keys
- To define compound primary key as follows

PRIMARY KEY (*partition_column_name*, *clustering_column_name* [, ...])

# CREATE TABLE

- Create the cyclist category table and store the data in reverse order:

```
CREATE TABLE cycling.cyclist_category (
   category text,
   points int,
   id UUID,
   lastname text,
   PRIMARY KEY (category, points))
WITH CLUSTERING ORDER BY (points DESC);
```

# CREATE INDEX

- Define a new index on a single column of a table

```
CREATE INDEX IF NOT EXISTS index_name
ON keyspace_name.table_name ( KEYS ( column_name ) )
```

- If data already exists for the column, Cassandra indexes the data during the execution of this statement
- After the index is created, Cassandra indexes new data for the column automatically when new data is inserted

# CREATE TABLE | CREATE INDEX

- Define a table and then create an index on two of its columns:

```
CREATE TABLE myschema.users (
   userID uuid,
   fname text,
   lname text,
   email text,
   address text,
   zip int,
   state text,
   PRIMARY KEY (userID)
 );

CREATE INDEX user_state ON myschema.users (state);

CREATE INDEX ON myschema.users (zip);
```

# INSERT

- Inserts an entire row or upserts data into an existing row, using the full primary key

```
INSERT INTO [keyspace_name.] table_name (column_list)
VALUES (column_values)
[IF NOT EXISTS]
[USING TTL seconds | TIMESTAMP epoch_in_microseconds]
```

- Requires a value for each component of the primary key, but not for any other columns
- Missing values are set to null
- INSERT returns no results unless IF NOT EXISTS is used.

# INSERT

- Insert a cyclist name using both a TTL and timestamp.

```
INSERT INTO cycling.cyclist_name (id, lastname, firstname)
  VALUES (6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47, 'KRUIKSWIJK','Steven')
  USING TTL 86400 AND TIMESTAMP 123456789;
```

Add IF NOT EXISTS to ensure that the operation is not done if a row with the same primary key already exists:

```
INSERT INTO cycling.cyclist_name (id, lastname, firstname)
   VALUES (c4b65263-fe58-4846-83e8-f0e1c13d518f, 'RATTO', 'Rissella')
IF NOT EXISTS;
```

# SELECT

- Returns one or more rows from a single Cassandra table
- Although a select statement without a where clause returns all rows from all partitions, it is not recommended

```
SELECT * | select_expression | DISTINCT partition
FROM [keyspace_name.] table_name
[WHERE partition_value
   [AND clustering_filters
   [AND static_filters]]]
[ORDER BY PK_column_name ASC|DESC]
[LIMIT N]
[ALLOW FILTERING]
```

# SELECT

- The FROM clause specifies the table to query
- You may want to precede the table name with the name of the keyspace followed by a period (.)
- If you do not specify a keyspace, Cassandra queries the current keyspace

- The following example returns the number of rows in the IndexInfo table in the system keyspace:

```
SELECT COUNT(*)
FROM system.IndexInfo;
```

# SELECT

- The LIMIT option sets the maximum number of rows that the query returns:

```
SELECT lastname
FROM cycling.cyclist_name
LIMIT 50000;
```

- Even if the query matches 105,291 rows, Cassandra only returns the first 50,000
- The cqlsh shell has a default row limit of 10,000
- The Cassandra server and native protocol do not limit the number of returned rows

# SELECT

- The WHERE clause introduces one or more relations that filter the rows returned by SELECT
- The column specification of the relation must be one of the following:
  - One or more members of the partition key of the table
  - A clustering column, only if the relation is preceded by other relations that specify all columns in the partition key
  - A column that is indexed using CREATE INDEX

# SELECT

- For example, the following table definition defines id as the table's partition key:

```
CREATE TABLE cycling.cyclist_career_teams ( id UUID
PRIMARY KEY, lastname text, teams set<text> );
```

- In this example, the SELECT statement includes in the partition key, so the WHERE clause can use the id column:

```
SELECT id, lastname, teams
FROM cycling.cyclist_career_teams
WHERE id=5b6962dd-3f90-4c93-8f61-eabfa4a803e2;
```

- Restriction: a relation that references the partition key can only use an equality operator (= or IN)

# SELECT

- Use a relation on a clustering column only if it is preceded by relations that reference all the elements of the partition key

```
CREATE TABLE cycling.cyclist_points (
   id UUID,
   firstname text,
   lastname text,
   race_title text,
   race_points int,
   PRIMARY KEY (id, race_points ));

SELECT sum(race_points)
FROM cycling.cyclist_points
WHERE id=e3b19ec4-774a-4d1c-9e5a-decec1e30aac
      AND race_points > 7;
```

# CQL Constraints

- CQL has many restrictions in comparison to SQL
- These restrictions prevent inefficient querying across a distributed database
- CQL queries should not visit a large number of nodes to retrieve required data
- This has the potential to impact cluster-wide performance

# CQL Constraints

- Thus CQL prevents the following…

- No arbitrary WHERE clause – Apache Cassandra prevents arbitrary predicates in a WHERE statement. Where clauses must have columns specified in your primary key.
- No JOINS – You cannot join data from two Apache Cassandra tables.
- No arbitrary GROUP BY – GROUP BY can only be applied to a partition or cluster column. Apache Cassandra 3.10 added GROUP BY support to SELECT statements.
- No arbitrary ORDER BY clauses – Order by can only be applied to a clustered column.

# Example

We need to build a system for an online electronic books reading site.

# Example

We need to build a system for an online electronic books reading site.

```
CREATE KEYSPACE e_library WITH REPLICATION =
{“class”:“SimpleStrategy”,“replication_factor”: 3};
```

# DDL

```
CREATE TABLE performer (  name
  VARCHAR,
  type VARCHAR,  country
  VARCHAR,  style
  VARCHAR,  founded INT,
  born TIMESTAMP,  died
  TIMESTAMP,  PRIMARY KEY
  (name)
);
```

# PRIMARY KEY

**PARTITION KEY + CLUSTERING COLUMN(S)**

# PRIMARY KEY

- Simple partition key, no clustering columns:
  - `PRIMARY KEY (name)`
- Composite partition key, no clustering columns:
  - `PRIMARY KEY ((album_title, year))`
- Simple partition key and clustering columns:
  - `PRIMARY KEY (album_title, number)`
- Composite partition key and clustering columns:
  - `PRIMARY KEY ((album_title, year), number)`

# PRIMARY KEYS

```
CREATE TABLE albums_by_track (
  track_titleVARCHAR,
  performerVARCHAR,
  year INT,
  album_titleVARCHAR,
  PRIMARY KEY (
    track_title, performer, year,album_title)
);
```

```
CREATE TABLE tracks_by_album (
  album_titleVARCHAR,
  year INT,
  performer VARCHAR STATIC,
  genre VARCHAR STATIC,
  number INT, track_title
  VARCHAR,
  PRIMARY KEY ((album_title, year), number)
);
```

| CQLTYPE | Constants | Description |
|---|---|---|
| ASCII | strings | US-ASCII character strings |
| BIGINT | integers | 64-bit signed long |
| BLOB | blobs | Arbitrary bytes (no validation), as hexadecimal |
| BOOLEAN | booleans | true or false |
| COUNTER | integers | Distributed counter value (64 bit long) |
| DECIMAL | integers or floats | Variable precision decimal |
| DOUBLE | integers | 64-bit IEEE-754 floating point |
| FLOAT | integers,floats | 32-bit IEEE-754 floating point |
| INT | integers | 32-bit signed integer |
| LIST | n/a | A collection of one or more ordered elements |
| MAP | n/a | A JSON style array of literals { literal: literal, literal: literal,…} |
| SET | n/a | A collection of one or more elements |
| TEXT | strings | UTF-8 encoded text |
| TIMESTAMP | integers,strings | Date + time as mills since EPOCH |
| UUID | uuids | Standard UUID |
| VARCHAR | strings | UTF-8 encoded string |
| VARINT | integers | Arbitrary precision integer |
| TIMEUUID | uuids | Type I UUID |

# INSERT

```
INSERT INTO albums_by_performer (performer, year,
title, genre) VALUES ('The Beatles', 1966, 'Revolver',
'Rock');
```

- CQL INSERTS are:
  - Atomic: Either all the values are inserted or none
  - Isolated: Two inserts on the exact same PK happen one after the other, no mixed values.

# UPDATE

```
UPDATE albums_by_performer
SET genre = 'Rock'
WHERE performer = 'The Beatles' AND
      year =  1966 AND
      title = 'Revolver';
```

- Primary Key columns cannot be changed.
- Full Primary key is required as predicate.
- CQL UPDATES are:
  - Atomic: Either all the values are inserted or none
  - Isolated: Two inserts on the exact same PK happen one after the other, no mixed values.

# UPSERT

```
INSERT INTO albums_by_performer (performer, year, title, genre)
VALUES ('The Beatles', 1966,'Revolver','Rock');
```

==

```
UPDATE albums_by_performer  SET
genre = 'Rock'
WHERE performer = 'The Beatles' AND  year
            =  1966 AND
            title = 'Revolver';
```

# Example

We need to design a system that holds users. Users will have name, ID card (unique), a phones list (home, mobile and work), birth date and an email address.

**NOTE**: As we haven't studied SELECT, use `SELECT * FROM <table name>;` to inspect your data.

# Example

```
CREATE TABLE users (
  ID VARCHAR PRIMARY KEY,
  name VARCHAR,  home_phone
  VARCHAR,  work_phone
  VARCHAR,  mobile_phone
  VARCHAR,  emailVARCHAR,
  birth_dateTIMESTAMP
);
```

# MORE DDL

```
ALTER TABLE album ADD cover_image VARCHAR;
```

```
ALTER TABLE album ALTER cover_image TYPE BLOB;
```

```
ALTER TABLE album DROP cover_image;
```

# MORE DDL

```
CREATE TABLE albums_by_genre (   genre
  VARCHAR,
  performerVARCHAR,  year INT,
  album_titleVARCHAR,  PRIMARY
  KEY (
    genre, performer, year,album_title)
) WITH CLUSTERING ORDER BY
  (performer ASC, year DESC,title ASC);
```

# SECONDARY INDEXES

- Tables are indexed on columns in a PK
  - Search on a partition key is very efficient
  - Search on a PK and Clustering column is very efficient
  - Search on other things is not supported
- Secondary indexes allow indexing other columns to be queried.
  - One index per column

# SECONDARY INDEXES

```
CREATE TABLE performer (  name VARCHAR,
  type VARCHAR,  country VARCHAR,
  style VARCHAR,  founded INT,
  born TIMESTAMP,  died TIMESTAMP,
  PRIMARY KEY (name)
);
```

```
CREATE INDEX performers_by_style  ON
 perfomer (style);
```

```
DROP INDEX performers_by_style;
```

# SECONDARY INDEXES

- Same recommendations for RDBMS
  - Use indexes on low cardinality fields
  - Beware of the write overhead
- Every node indexes it local data therefore => a read hits all nodes!!
  - Don't use them. Use lookup tables instead.

# Example

We need to query the users by name.

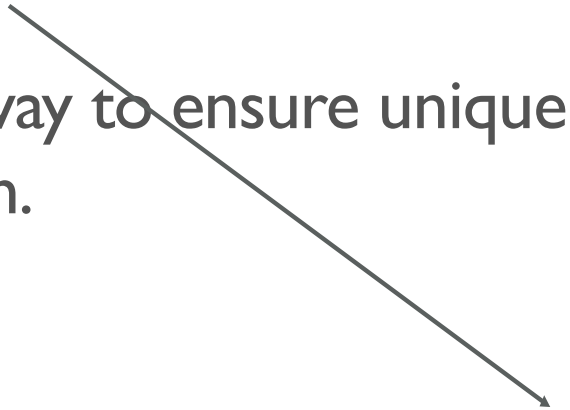
# Example

We need to query the users by name.

```
CREATE INDEX users_by_name  ON
 users (name);
```

# UUID

- Type 4 UUID
- Our way to ensure uniqueness in a distributed system.



**7ffa4040-9132-4e0b-b04f-610e869d8717**

# Example

Our system has another entity: Books. Books have a title and an author. We have no guarantee of any of them or even their combination to be unique.

# Example

Our system has another entity: Books. Books have a title and an author. We have no guarantee of any of them or even their combination to be unique.

```
CREATE TABLE books (
  uid TIMEUUID PRIMARY KEY,
  title VARCHAR,
  author
  VARCHAR
);
```

# TIMEUUID

- Timestamp + UUID

**c9cc9e60-711c-11e5-9d70-feff819cdc9f**

- Type 1 UUID
- Generated with CQL now() function
- Can extract the Timestamp with CQL dateof() function

# TIMEUUID

- Timestamp + UUID
- Type 1 UUID


- Generated with CQL now() function
- Can extract the Timestamp with CQL dateof() function

# TIMEUUID

```
CREATE TABLE track_ratings_by_user (
user UUID,
activity TIMEUUID,
rating INT,
album_title VARCHAR,
album_year INT, track_title VARCHAR,
PRIMARY KEY (user, activity)
) WITH CLUSTERING ORDER (activity DESC);
```

# TTL

```
INSERT INTO albums_by_performer (performer, year, title,
genre) VALUES ('The Beatles', 1966,'Revolver','Rock')
USING TTL 30;
```

- Time To Live for columns specified in seconds.
- After TTL expires, column is marked with a Tombstone.

# Example

We are in the BigData era and therefore we want to measure absolutely everything our users do in our portal. Actions will be defined by a type (string) and a receiver (int).

# Example

We are in the BigData era and therefore we want to measure absolutely everything our users do in our portal. Actions will be defined by a type (string) and a receiver (int).

```
CREATE TABLE user_action (
  user_ID VARCHAR,
  time TIMESTAMP,
  type VARCHAR,
  receiver INT,
  PRIMARY KEY(user_ID, time)
);
```

# DELETE

- A whole partition:
  - `DELETE FROM <table> WHERE <partition_key> = value;`
- A row:
  - `DELETE FROM <table> WHERE <primary key> = value;`
- A column:
  - `DELETE <column name> FROM <table> WHERE <primary key> = value;`
- Deleted things are marked with a tombstone, not actually removed.

# TRUNCATE

```
TRUNCATE albums_by_performer;
```

# TIMEUUID

- Timestamp + UUID
- Type 1 UUID


- Generated with CQL now() function
- Can extract the Timestamp with CQL dateof() function

# COLLECTIONS

**Our users can have several email addresses…**

- Set: Uniqueness
  - `email_addresses SET<VARCHAR>`
- List: Order
  - `email_addresses LIST<VARCHAR>`
- Map: Key-Value pairs
  - `email_addresses MAP<VARCHAR, VARCHAR>`

# SETS

- Insert:
  - `INSERT INTO band (name, members) VALUES ('The Beatles', {'John', 'Paul', 'George'});`
- Union (duplicates deletion managed transparently):
  - `UPDATE band SET members = members + {'John', 'Ringo'} WHERE name = 'The Beatles';`
- Difference:
  - `UPDATE band SET members = members - {'Ringo'} WHERE name = 'The Beatles';`
- Deletion:
  - `DELETE members FROM band WHERE name = 'The Beatles';`

# LISTS

```
CREATE TABLE song (
  name VARCHAR
  songwriters LIST<VARCHAR>,
  PRIMARY KEY (name)
);
```

- Insert:
  - `INSERT INTO song (name, songwriters) VALUES ('Hold your hand', ['John', 'Paul']);`
- Append:
  - `UPDATE song SET songwriters = songwriters +['Paul'] WHERE name = …;`

# LISTS

- Prepend:
  - `UPDATE song SET songwriters = ['Paul'] + songwriters WHERE name = …;;`
- Update:
  - `UPDATE song SET songwriters[1] = 'Jonathan' WHERE name = …;`
- Subtract
  - `UPDATE song SET songwriters = songwriters - ['Jonathan'] WHERE name = …;`
- Delete
  - `DELETE songwriters[0] FROM song WHERE name = …;`

# MAPS

- Insert:
  - `INSERT INTO album (title, tracks) VALUES ('Revolver', { 1: 'Taxman', 2: 'Eleanor'});`
- Update:
  - `UPDATE album SET tracks[3] = 'Yellow Submarine' WHERE title = …;`
- Delete:
  - `DELETE tracks[3] FROM album WHERE title = …;`

```
CREATE TABLE album (
titleVARCHAR,
 tracks
MAP<INT,VARCHAR>,
PRIMARY KEY (title)
);
```

# Example

Our users can define a set of preferences in the portal: TimeZone, Language and Currency

# Example

Our users can define a set of preferences in the portal: TimeZone, Language and Currency

```
ALTER TABLE users ADD preferences
MAP<VARCHAR,VARCHAR>;
```

# SELECT

- All rows:
  - `SELECT * FROM album;`
- Specific columns:
  - `SELECT performer, title, year FROM album;`
- Specific field from a UDT:
  - `SELECT performer.lastname FROM album;`
- Count:
  - `SELECT COUNT(*) FROM album;`

# WHERE

- Equality matches:
  - `SELECT * FROM tracks_by_album WHERE album_title = 'Revolver' AND year = 1966;`
  - `SELECT * FROM tracks_by_album WHERE album_title = 'Revolver' AND year = 1966 AND number = 6;`
- IN:
  - Only applicable in the last WHERE clause
  - `SELECT * FROM tracks_by_album WHERE album_title = 'Revolver' AND year = 1966 AND number IN (2, 3, 4);`

# WHERE

- Range search:
  - Only on clustering columns.
  - `SELECT * FROM tracks_by_album WHERE album_title = 'Revolver' AND year = 1966 AND` **`number >= 6 AND number < 2;`**
- ALLOW FILTERING:
  - Allows scanning through all partitions => potentially very time consuming
  - `SELECT * FROM tracks_by_album WHERE number = 2 ALLOW FILTERING;`

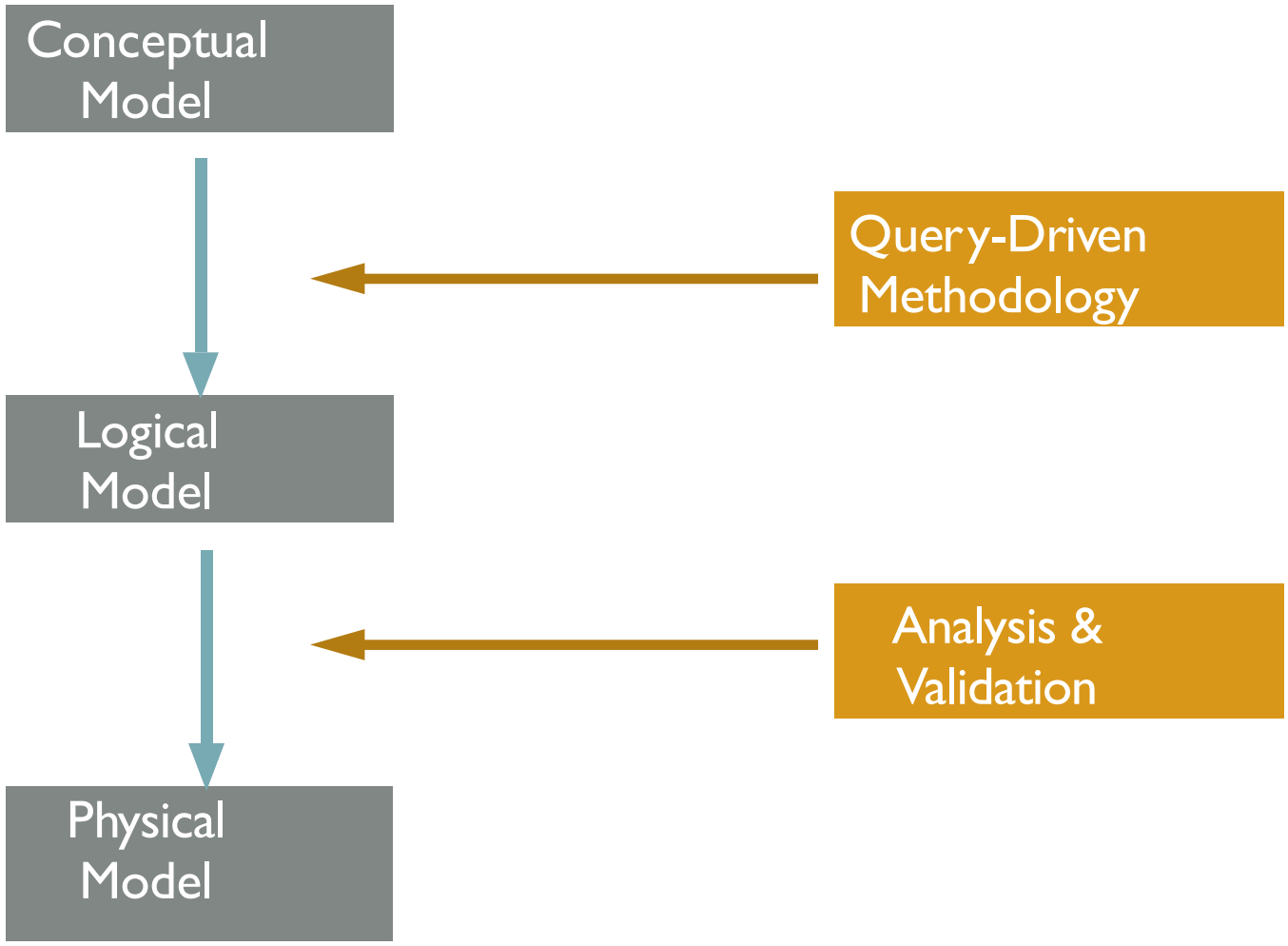
# DATA MODELLING

Processes and good practices to design our schema.

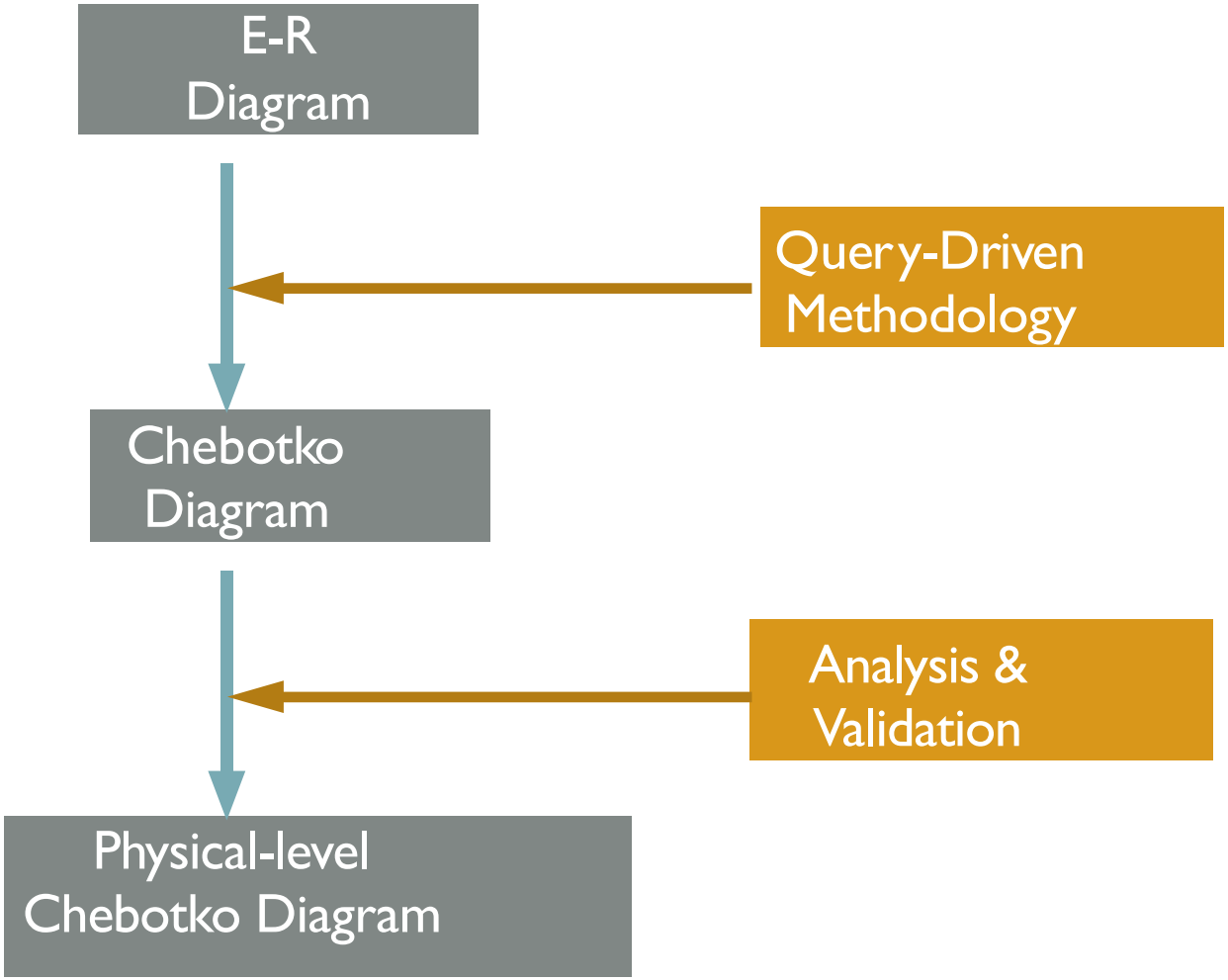
# DATA MODELLING

- Understand your data
- Decide how you'll query the data
- Define column families to satisfy those queries
- Implement and optimize

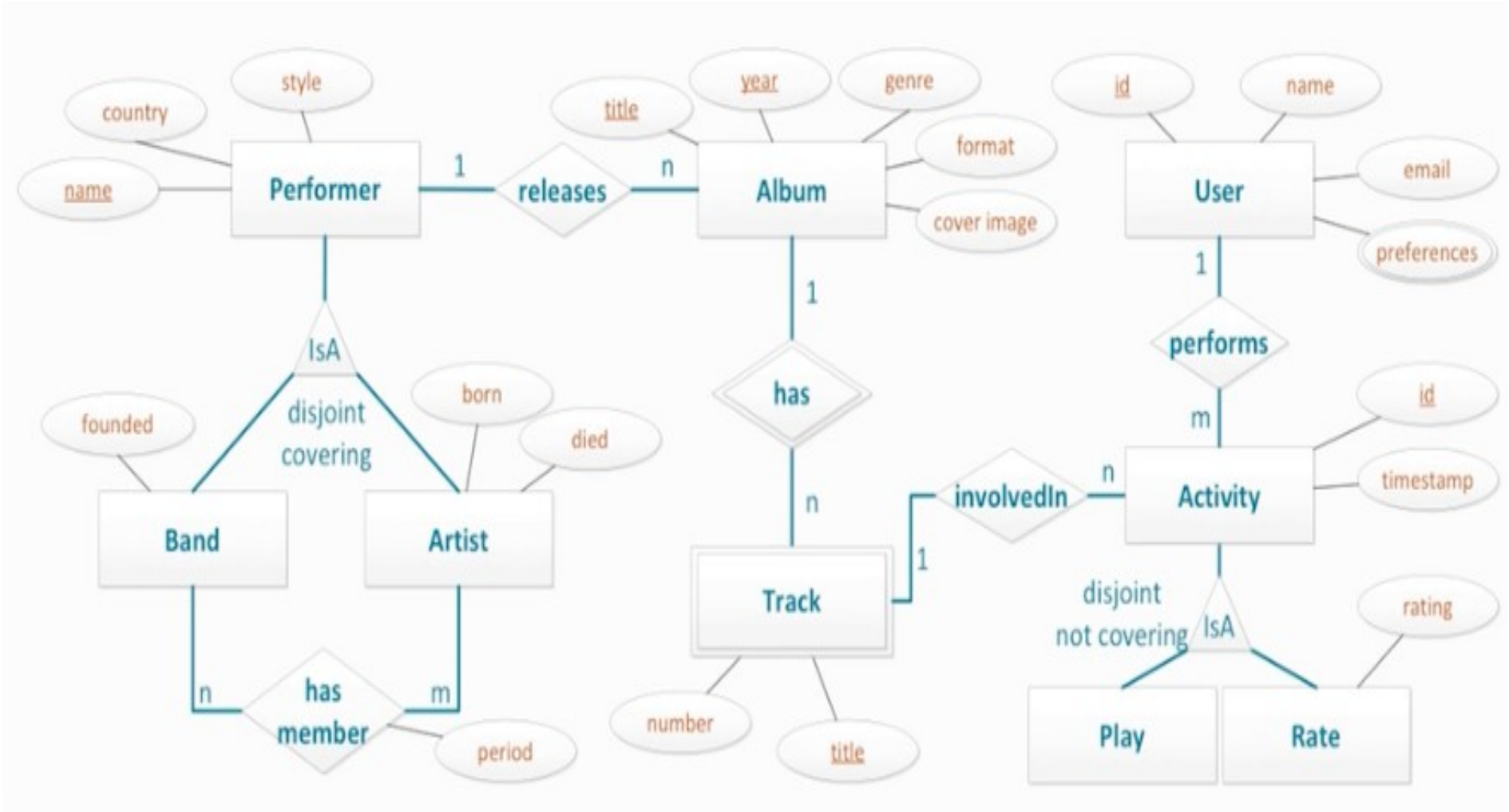
# DATA MODELLING



# DATA MODELLING



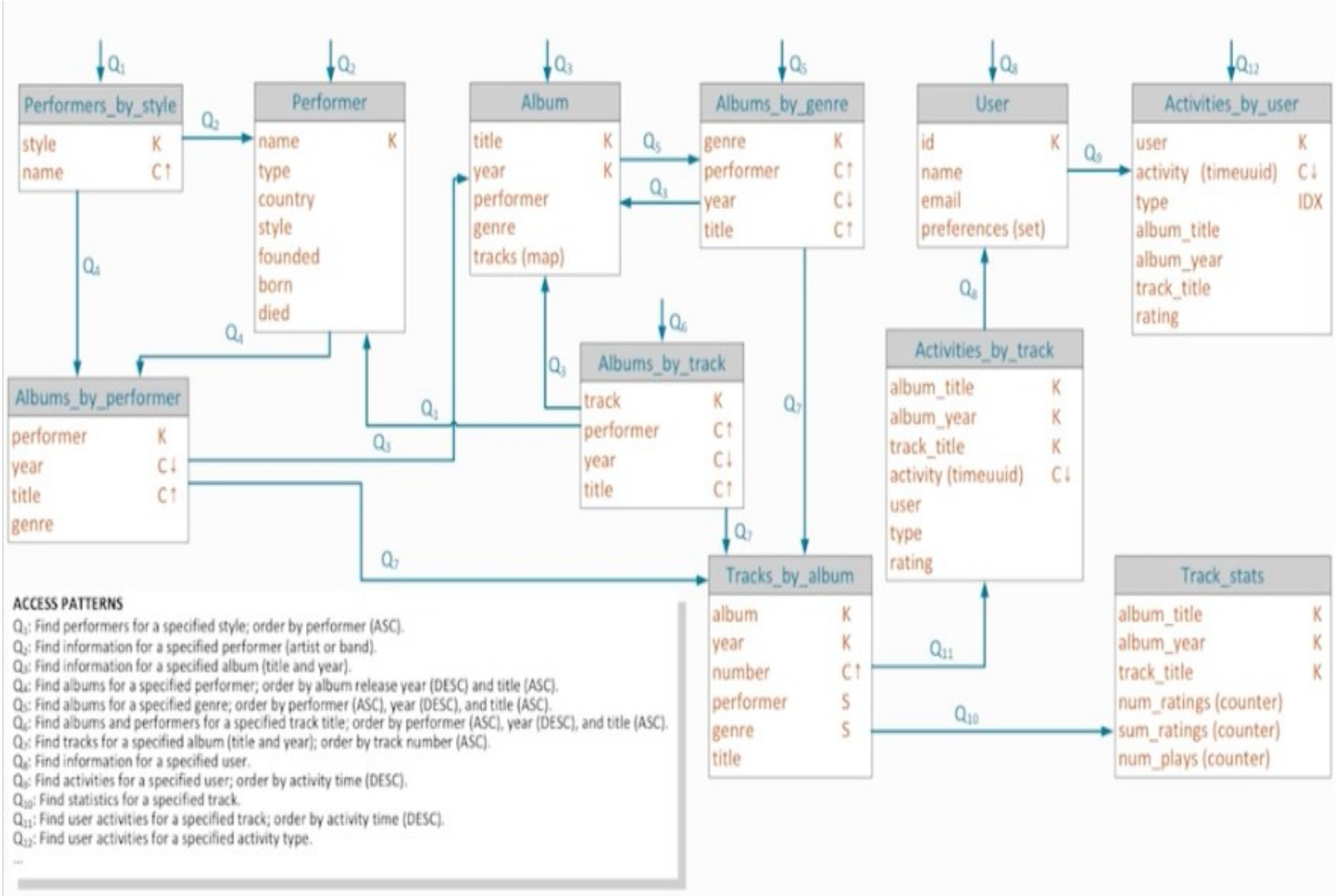
# CONCEPTUAL MODEL



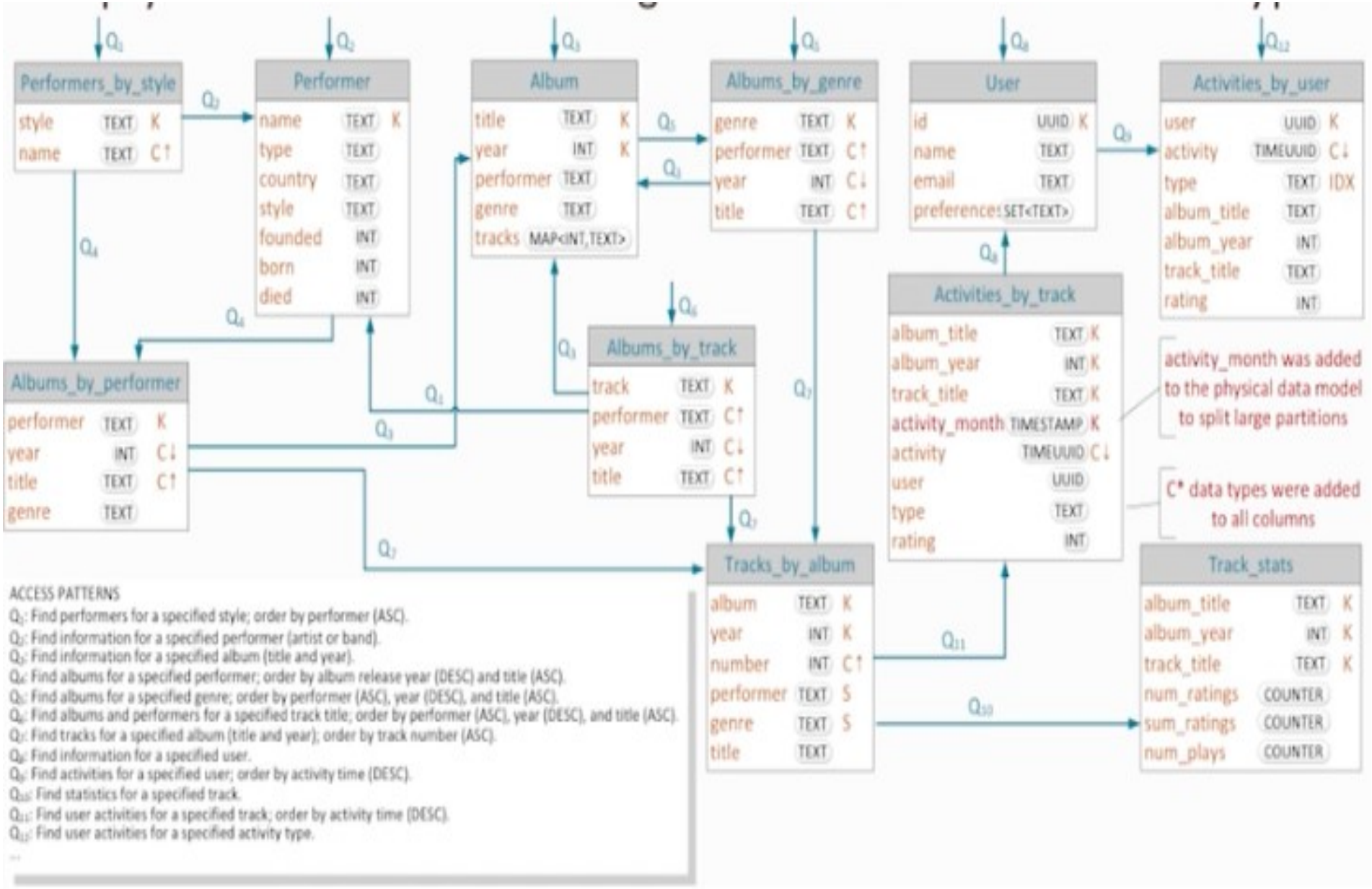
# QUERY DRIVEN METHODOLOGY

- Spread data evenly around the cluster
- Minimize the number of partitions read
- Follow the mapping rules:
  - Entities and relationships: map to tables
  - Equality search attributes: must be at the beginning of the primary key
  - Inequality search attributes: become clustering columns
  - Ordering attributes: become clustering columns
  - Key attributes: map to primary key columns

# LOGICAL


Performers_by_style
style K
name C↑
Performer
name K
type
country
style
founded
born
died
Album
title K
year K
performer
genre
tracks (map)
Albums_by_genre
genre K
performer C↑
year C↓
title C↑
User
id K
name
email
preferences (set)
Activities_by_user
user K
activity (timeuuid) C↓
type IDX
album_title
album_year
track_title
rating
Albums_by_performer
performer K
year C↓
title C↑
genre
Albums_by_track
track K
performer C↑
year C↓
title C↑
Activities_by_track
album_title K
album_year K
track_title K
activity (timeuuid) C↓
user
type
rating
Tracks_by_album
album K
year K
number C↑
performer S
genre S
title
Track_stats
album_title K
album_year K
track_title K
num_ratings (counter)
sum_ratings (counter)
num_plays (counter)
ACCESS PATTERNS
Q1: Find performers for a specified style; order by performer (ASC).
Q2: Find information for a specified performer (artist or band).
Q3: Find information for a specified album (title and year).
Q4: Find albums for a specified performer; order by album release year (DESC) and title (ASC).
Q5: Find albums for a specified genre; order by performer (ASC), year (DESC), and title (ASC).
Q6: Find albums and performers for a specified track title; order by performer (ASC), year (DESC), and title (ASC).
Q7: Find tracks for a specified album (title and year); order by track number (ASC).
Q8: Find information for a specified user.
Q9: Find activities for a specified user; order by activity time (DESC).
Q10: Find statistics for a specified track.
Q11: Find user activities for a specified track; order by activity time (DESC).
Q12: Find user activities for a specified activity type.

# PHYSICAL


Performers_by_style
style TEXT K
name TEXT C↑
Performer
name TEXT K
type TEXT
country TEXT
style TEXT
founded INT
born INT
died INT
Album
title TEXT K
year INT K
performer TEXT
genre TEXT
tracks MAP<INT,TEXT>
Albums_by_genre
genre TEXT K
performer TEXT C↑
year INT C↓
title TEXT C↑
User
id UUID K
name TEXT
email TEXT
preferences SET<TEXT>
Activities_by_user
user UUID K
activity TIMEUUID C↓
type TEXT IDX
album_title TEXT
album_year INT
track_title TEXT
rating INT
Albums_by_performer
performer TEXT K
year INT C↓
title TEXT C↑
genre TEXT
Albums_by_track
track TEXT K
performer TEXT C↑
year INT C↓
title TEXT C↑
Activities_by_track
album_title TEXT K
album_year INT K
track_title TEXT K
activity_month TIMESTAMP K
activity TIMEUUID C↓
user UUID
type TEXT
rating INT
activity_month was added to the physical data model to split large partitions
C* data types were added to all columns
Tracks_by_album
album TEXT K
year INT K
number INT C↑
performer TEXT S
genre TEXT S
title TEXT
Track_stats
album_title TEXT K
album_year INT K
track_title TEXT K
num_ratings COUNTER
sum_ratings COUNTER
num_plays COUNTER
ACCESS PATTERNS
Q1: Find performers for a specified style; order by performer (ASC).
Q2: Find information for a specified performer (artist or band).
Q3: Find information for a specified album (title and year).
Q4: Find albums for a specified performer; order by album release year (DESC) and title (ASC).
Q5: Find albums for a specified genre; order by performer (ASC), year (DESC), and title (ASC).
Q6: Find albums and performers for a specified track title; order by performer (ASC), year (DESC), and title (ASC).
Q7: Find tracks for a specified album (title and year); order by track number (ASC).
Q8: Find information for a specified user.
Q9: Find activities for a specified user; order by activity time (DESC).
Q10: Find statistics for a specified track.
Q11: Find user activities for a specified track; order by activity time (DESC).
Q12: Find user activities for a specified activity type.