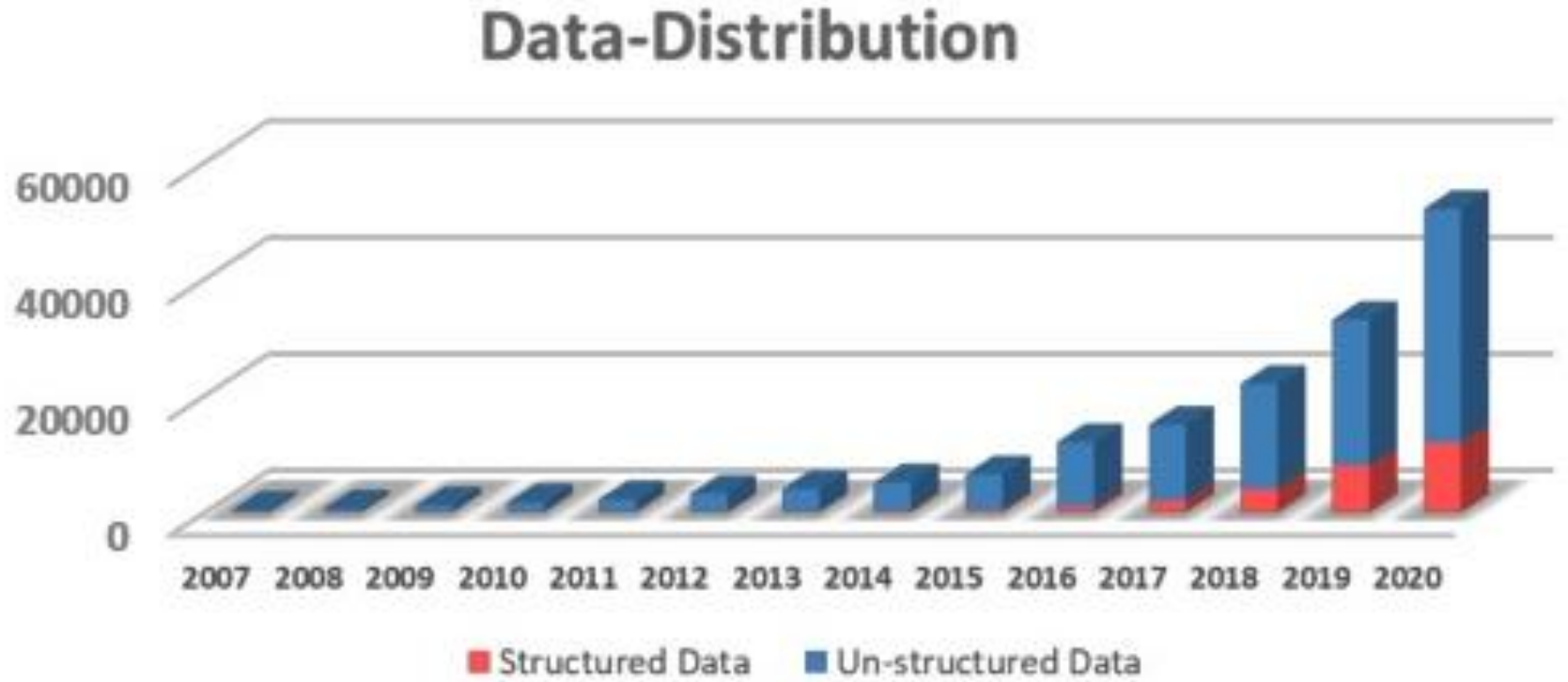
Cassandra

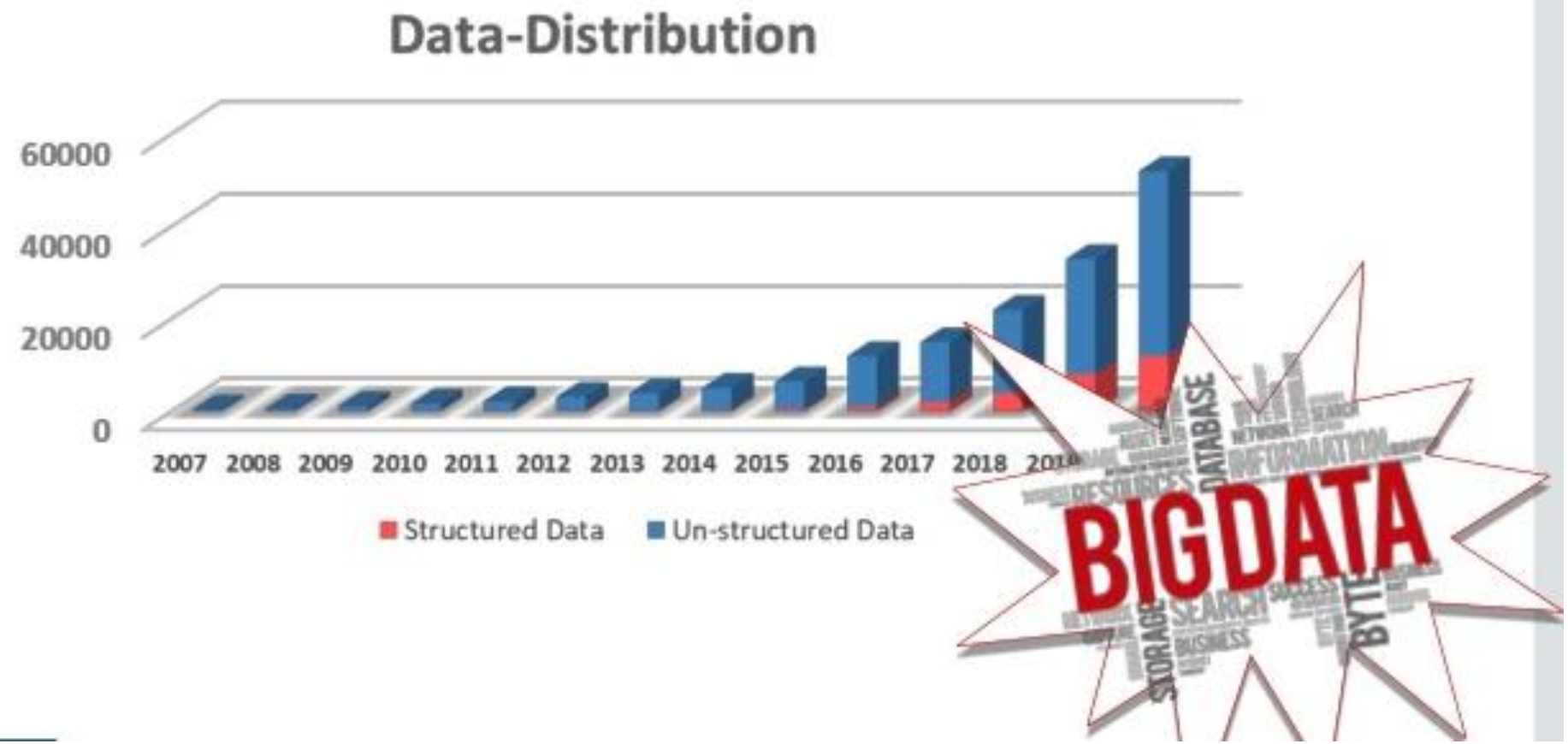
ANJU MUNOTH

Structure of Data Today

Structure of Data Today



Structure of Data Today



Big Data


INSTAGRAM
USERS LIKE
1,736,111
POSTS
YOUTUBE
USERS UPLOAD
300 HOURS
OF NEW VIDEO
every
MINUTE
of the DAY
Facebook
USERS LIKE
4,166,667
POSTS
REDDIT
USERS CAST
18,327
VOTES
TWITTER
USERS SEND
347,222
TWEETS

5 V's of Big Data

Volume

Processing increasing huge data sets

Variety

Processing different types of data

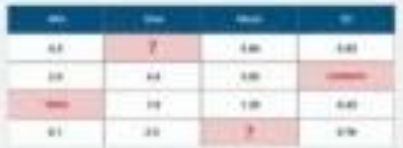
Velocity

Data is being generated at an alarming rate

Value

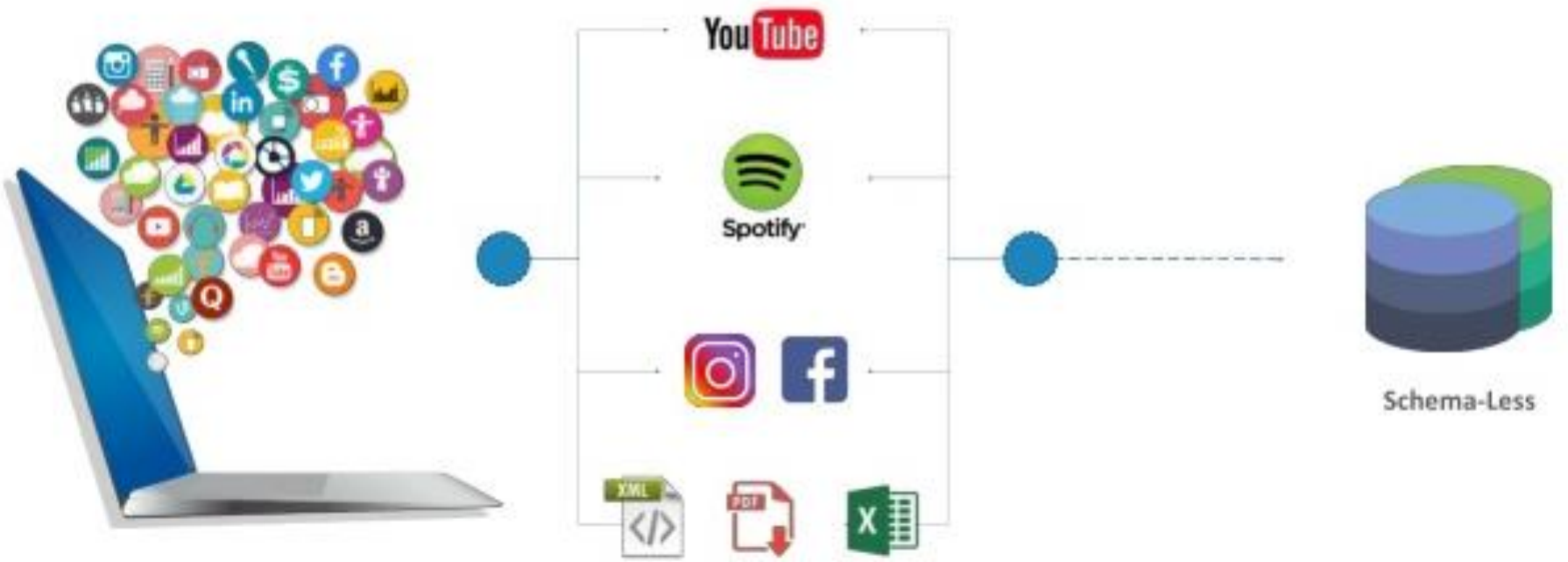
Finding correct meaning out of the data

Veracity

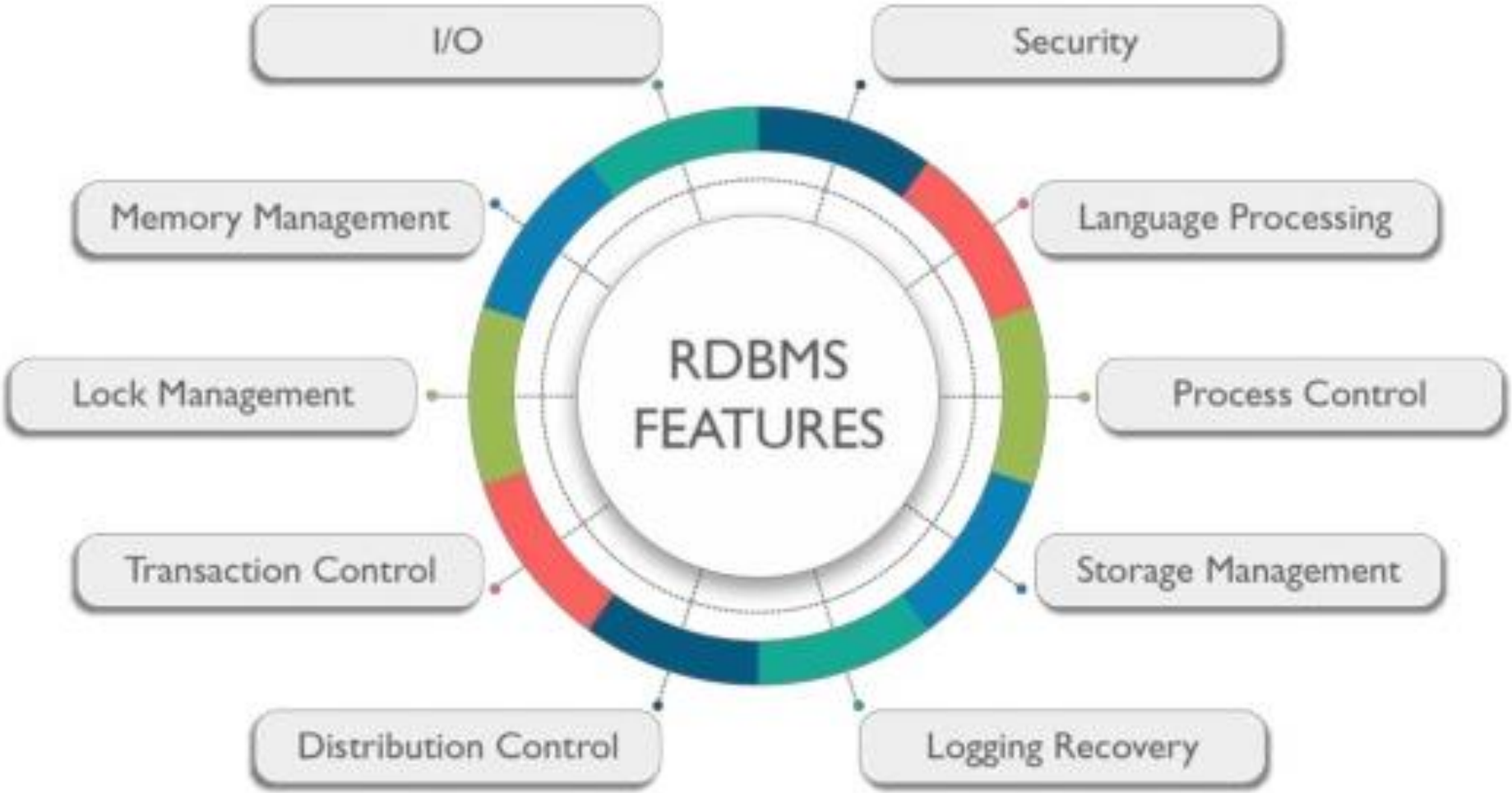


Uncertainty and inconsistencies in the data

Numerous Sources of Big Data

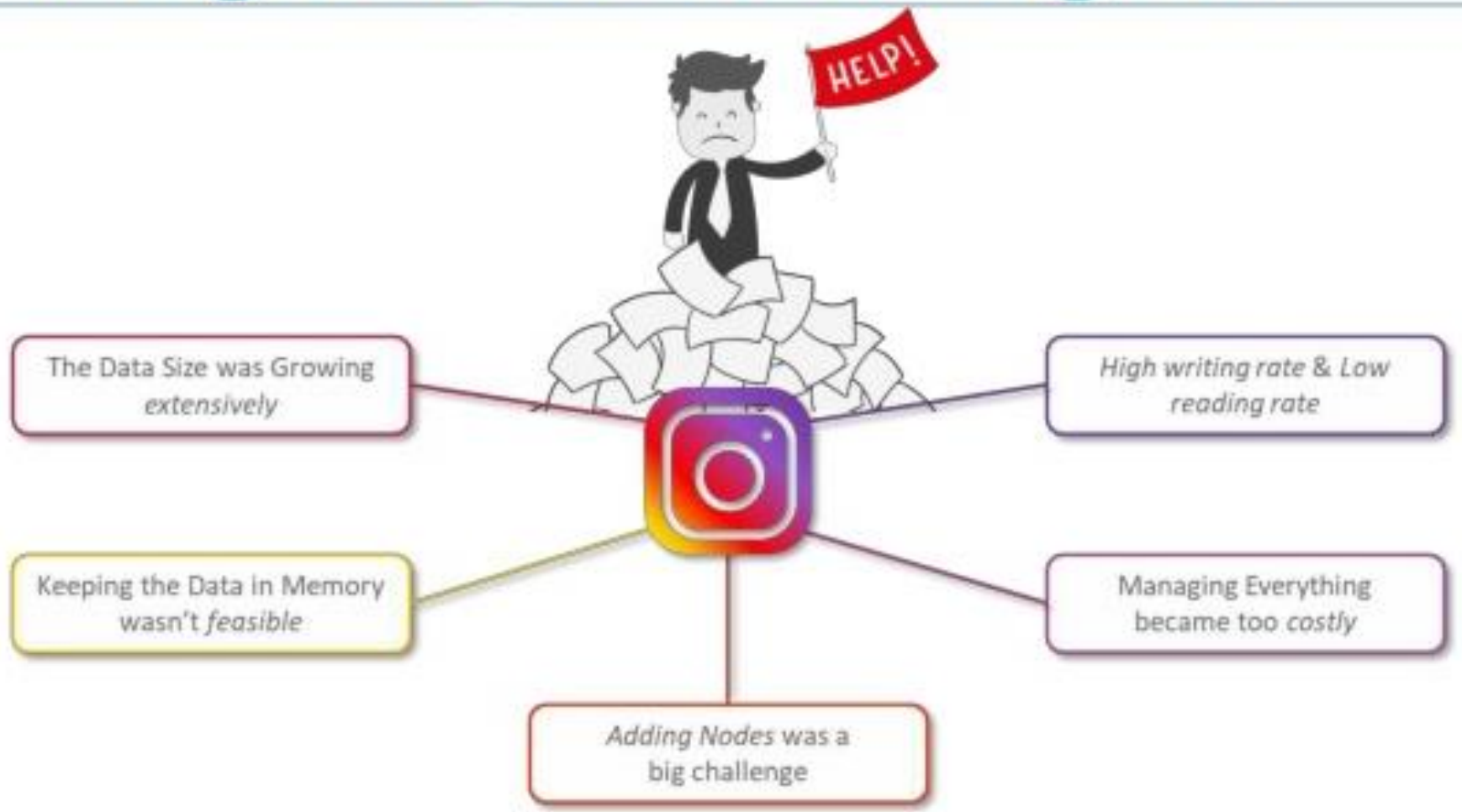


Traditional RDBMS



RDBMS Failing

Challenges faced at Instagram




SCALABLE


SCALABLE
COST
EFFECTIVE


SCALABLE
COST EFFECTIVE
HIGHLY AVAILABLE


SCALABLE
COST EFFECTIVE
HIGHLY AVAILABLE
FAST PROCESSING

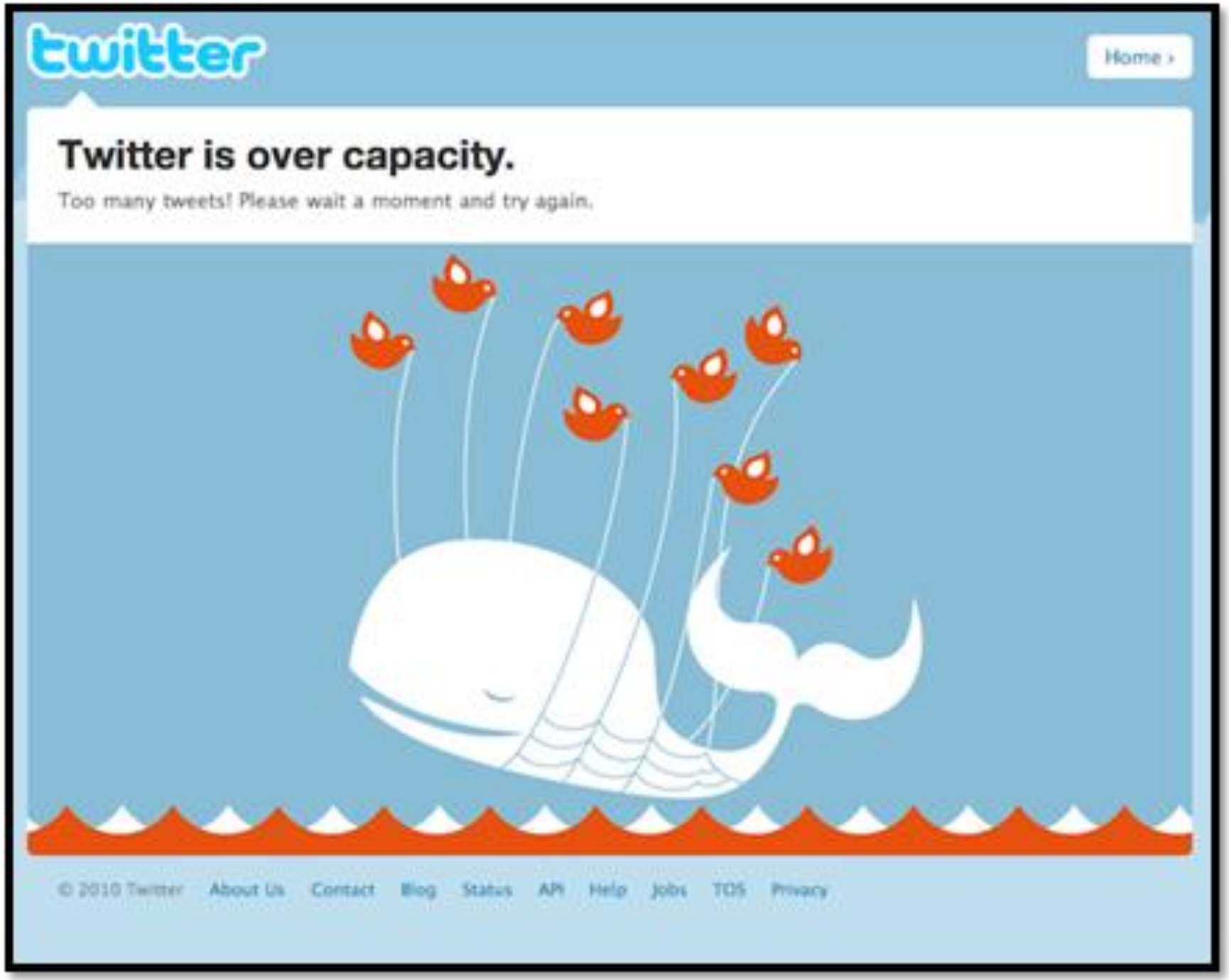

SCALABLE
SHOULD NOT RUN OUT OF MEMORY
COST EFFECTIVE
HIGHLY AVAILABLE
FAST PROCESSING

NEED OF NOSQL

- Modern RDBMS simply don't scale to internet traffic. So, the main solutions are-
  - Scaling Up(Vertical Scaling)- Adding resources to a single node in a system.
  - Scaling Out(Horizontal Scaling)- Adding more nodes to a single system, multi-node database solution. Different approaches are-
    - Master Slave
    - Sharding
    - Multi Master Replication
    - In Memory database
    - No Joins

Need of NOSQL(cont.)

- High prices from RDBMS vendors.
- These days Sites like Digg, Facebook and EBay have data sets 10s or 100s of TB large.
- Reduce object-relational impedance


twitter
Home ›
Twitter is over capacity.
Too many tweets! Please wait a moment and try again.
© 2010 Twitter
About Us
Contact
Blog
Status
API
Help
Jobs
TOS
Privacy

Typical RDBMS Implementations

- Fixed table schemas
- Small but frequent reads/writes
- Large batch transactions
- Focus on ACID
  - Atomicity
  - Consistency
  - Isolation
  - Durability

Big Data Definition

- Volumes & volumes of data
- Unstructured
- Semi-structured
- Not suited for Relational Databases
- Often utilizes MapReduce frameworks

Big Data Examples

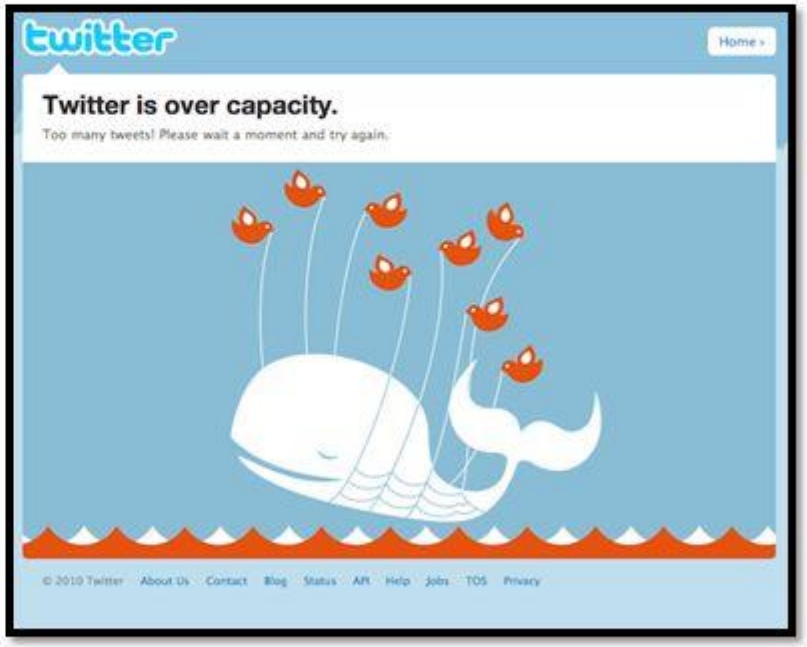
- Cassandra
- Hadoop
- Greenplum
- Azure Storage
- EMC Atmos
- Amazon S3
- SQL Azure (with Federations support)

Real World Example

- Twitter
  - The challenges
    - Needs to store many graphs
      - Who you are following
      - Who's following you
      - Who you receive phone notifications from etc
    - To deliver a tweet requires rapid paging of followers
    - Heavy write load as followers are added and removed
    - Set arithmetic for @mentions (intersection of users).

What did they try?

- Started with Relational Databases
- Tried Key-Value storage of denormalized lists
- Did it work?
  - Nope
    - Either good at
      - Handling the write load
      - Or paging large amounts of data
      - But not both

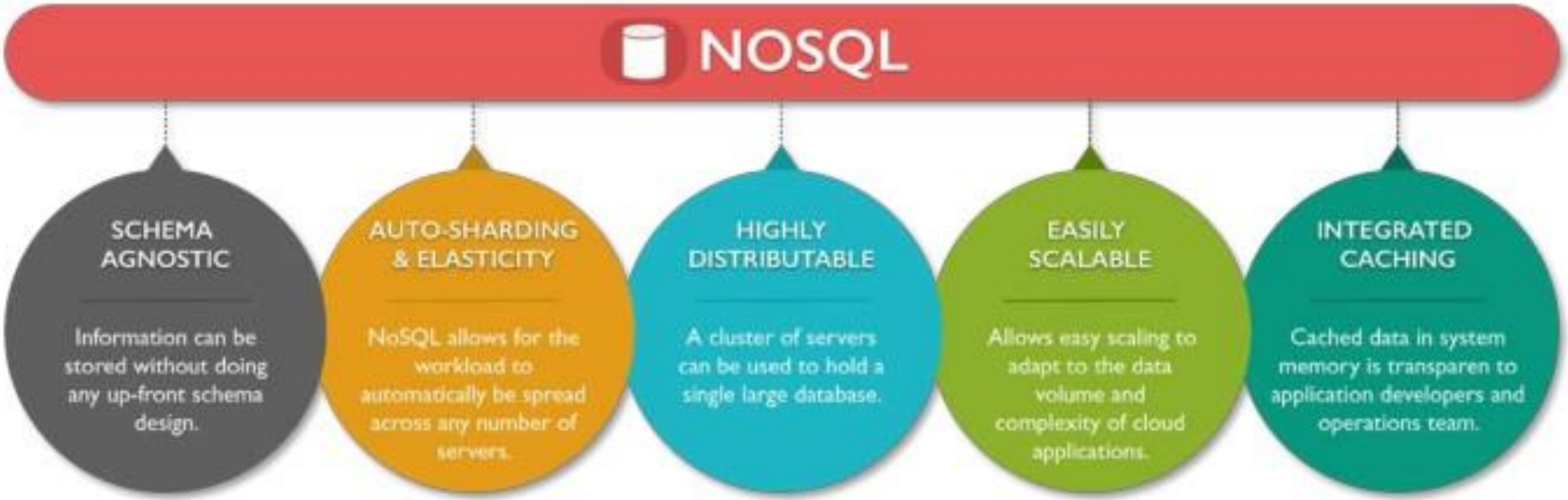


What did they need?

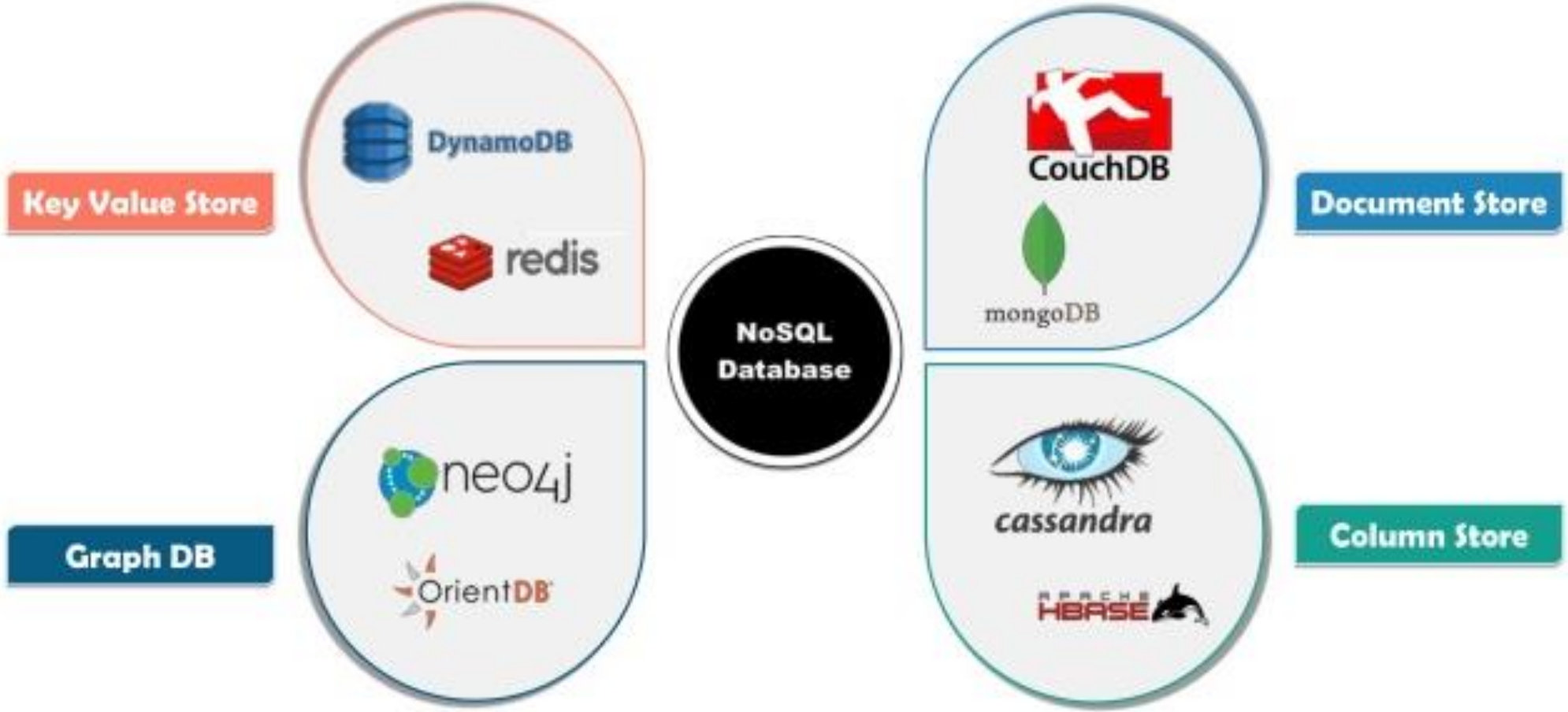
- Simplest possible thing that would work
- Allow for horizontal partitioning
- Allow write operations to
- Arrive out of order
  - Or be processed more than once
  - Failures should result in redundant work
- Not lost work!


NoSQL to the Rescue

NoSQL


NOSQL
SCHEMA AGNOSTIC
Information can be stored without doing any up-front schema design.
AUTO-SHARDING & ELASTICITY
NoSQL allows for the workload to automatically be spread across any number of servers.
HIGHLY DISTRIBUTABLE
A cluster of servers can be used to hold a single large database.
EASILY SCALABLE
Allows easy scaling to adapt to the data volume and complexity of cloud applications.
INTEGRATED CACHING
Cached data in system memory is transparen to application developers and operations team.

Types of NoSQL Databases



HISTORY OF NOSQL

- CARLO STROZZI used the term NOSQL in 1998 to name his lightweight, Open Source Relational Database.

- ERIC EVANS, a rack space employee, reintroduced the term NOSQL in 2009 to discuss the open source distributed database.

INTRODUCTION

- Stands for Not Only SQL.
- Having non-relational flat file database.
- May not require fixed table schema.
- Horizontally scalable-easily add more information.
- Avoid JOIN operation.
- Relaxation of ACID properties.
- Distributed in nature.

ARCHITECTURE of NOSQL

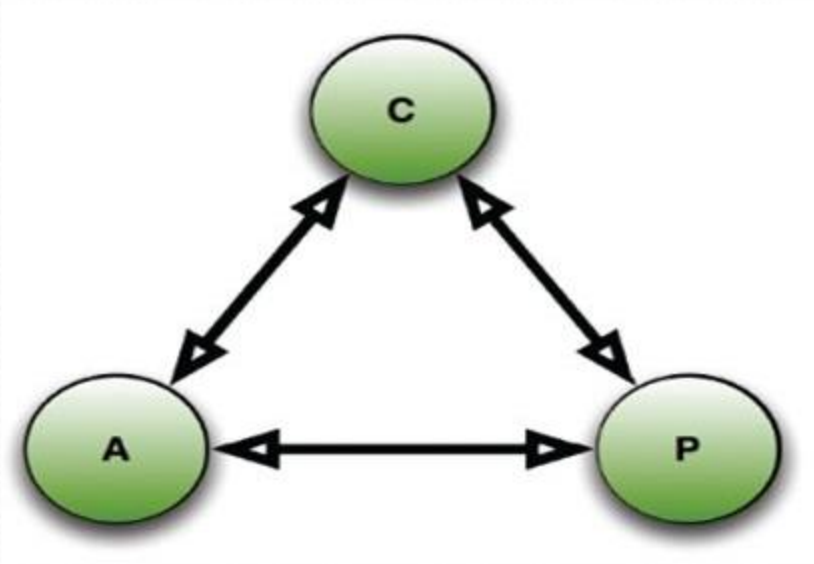
- Several NOSQL systems use a Distributed architecture like Master/Slave, Master/Master,Masterless.
- Based on Distributed Hash Tables.
- Easily scale out by adding more servers & failure of a server can be tolerated.
- Provide full ACID guarantee by adding a supplementary middleware layer.

CAP THEOREM

- Main movement of NOSQL- Having three main properties of a system:
  - Consistency
  - Availability
  - Partitions
- To scale out, you have to Partition. That leaves either Consistency or Availability , you would choose availability over consistency in almost all cases.
- You can have at most two of these three properties for any shared-data system.

CAP THEOREM(cont.)

Theorem states: Strict Consistency can't be achieved at the same time as availability and partition-tolerance.



What is CAP?

According to the CAP theorem it is not possible for a distributed data store to provide more than two of the following guarantees simultaneously.

- Consistency: Consistency implies that every read receives the most recent write or errors out
- Availability: Availability implies that every request receives a response. It is not guaranteed that the response contains the most recent write or data.
- Partition tolerance: Partition tolerance refers to the tolerance of a storage system to failure of a network partition. Even if some of the messages are dropped or delayed the system continues to operate.
- CAP theorem implies that when using a network partition, with the inherent risk of partition failure, one has to choose between consistency and availability and both cannot be guaranteed at the same time

EVENTUAL CONSISTENCY

- NOSQL database leaves consistency to achieve better availability & partitioning, this resulted in system know as **BASE.**
- **BASE ,** as opposed to ACID, having three properties:
  - **B**asically **A**vailable
  - **S**oft-state
  - **E**ventually consistent
- When no updates occur for a long period of time, eventually all updates will propagate through the system and all the nodes will be consistent.

NOSQL DATA MODELS

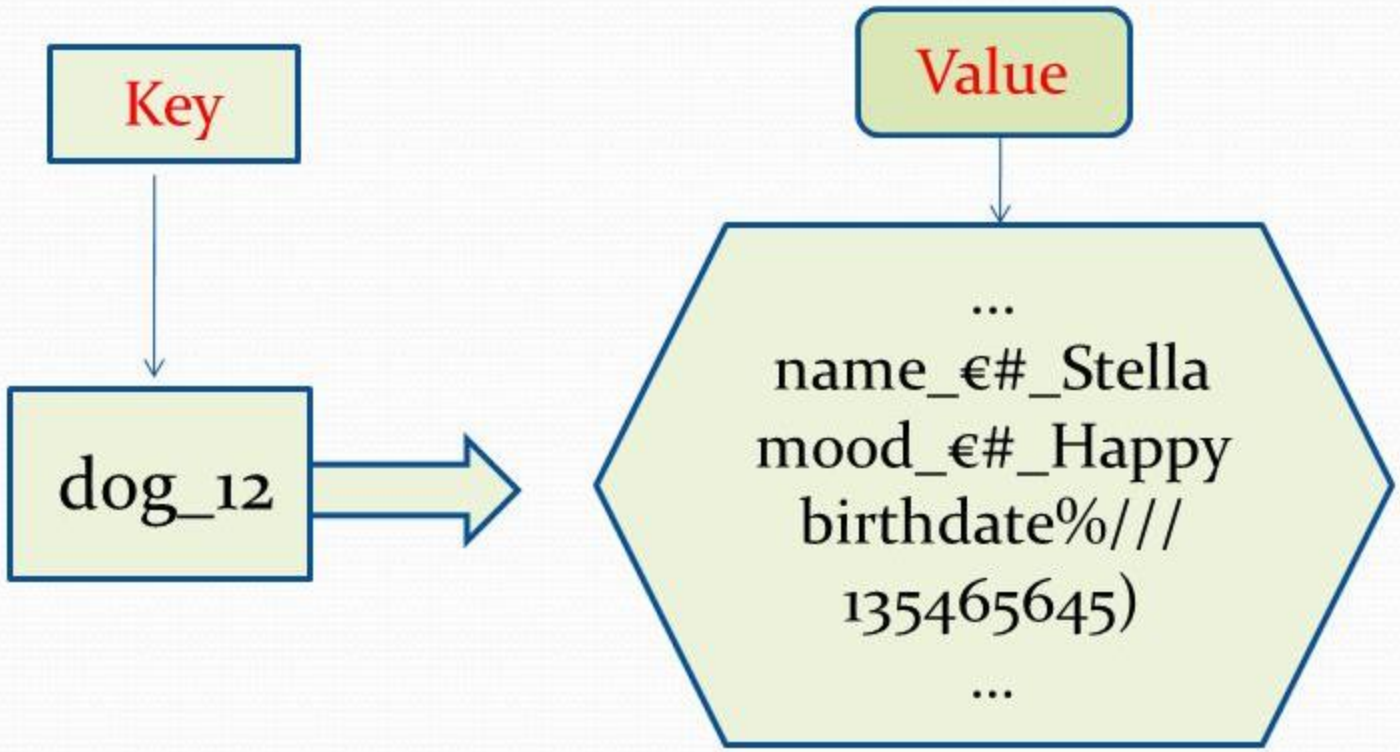
NOSQL databases can be categorized according to their data model into the following four categories:

- Key-Value-stores
- Document-stores
- Graph Databases
- Big Table-Column Implementation

KEY VALUE STORE

- Simplest form of NoSQL store- Each key is mapped to a value containing arbitrary data.
- This store has no knowledge of the contents of its payload and simply delivers the data to the application.
- Mainly used to encapsulate the information .
- Key-value stores is a very simple query model, usually consisting of set, get, and delete primitives.
- Main applications based on this store:
  - Redis
  - Level DB
  - Memcache DB

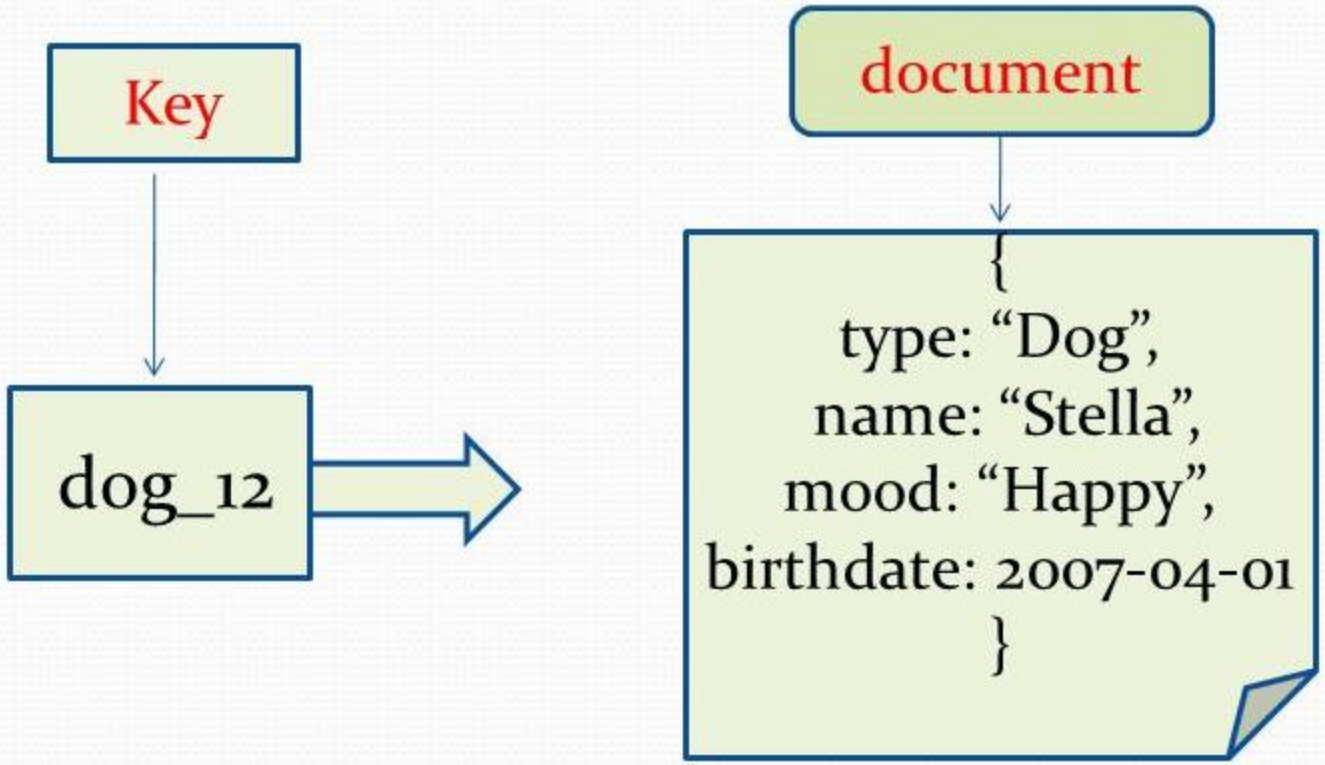
Key-value stores



DOCUMENT STORE

- Key-document stores map a key to some document that contains structured information.
- They store lists and dictionaries, which can be embedded recursively inside one-another.
- Freedom and complexity of document stores are two key points:
  - Developers have a lot of freedom in modeling their documents
  - Application-based query logic can become complex.
- Main applications based on this store:
    - Mongo DB
    - Couch DB
    - Riak

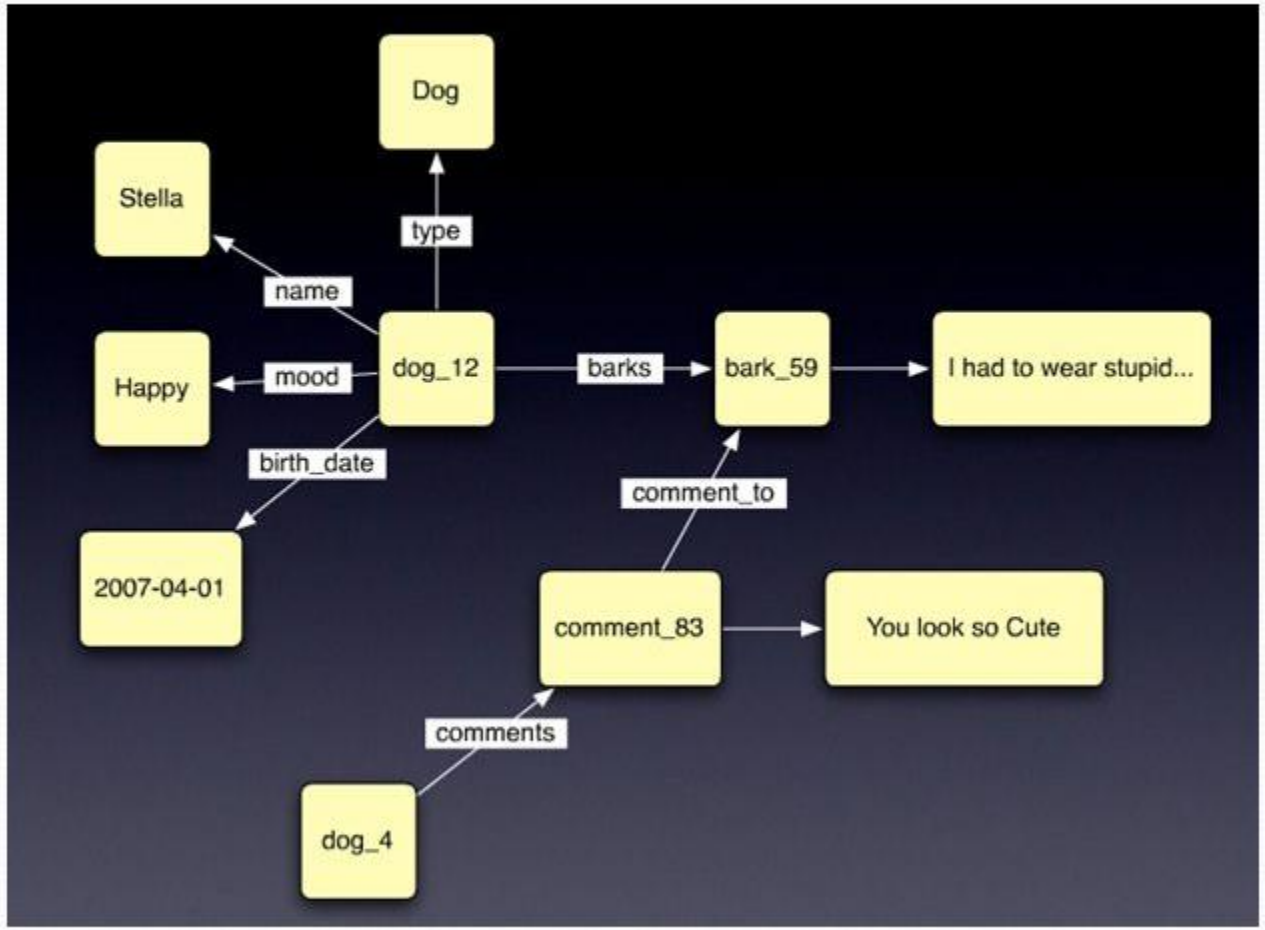
Document databases



GRAPH DATABASE

- To avoid JOIN operation in RDBMS, Graph Databases are used.
- Graph Database is modeled using three basic building blocks:
  - **Node** as vertex
  - **Relationship** as edge
  - **Property** as attribute
- Graph theory has seen a great usefulness and relevance in many problems across various domains.
- Main applications based on this store:
  - Neo4J
  - InfoGrid
  - HyperGraphDB

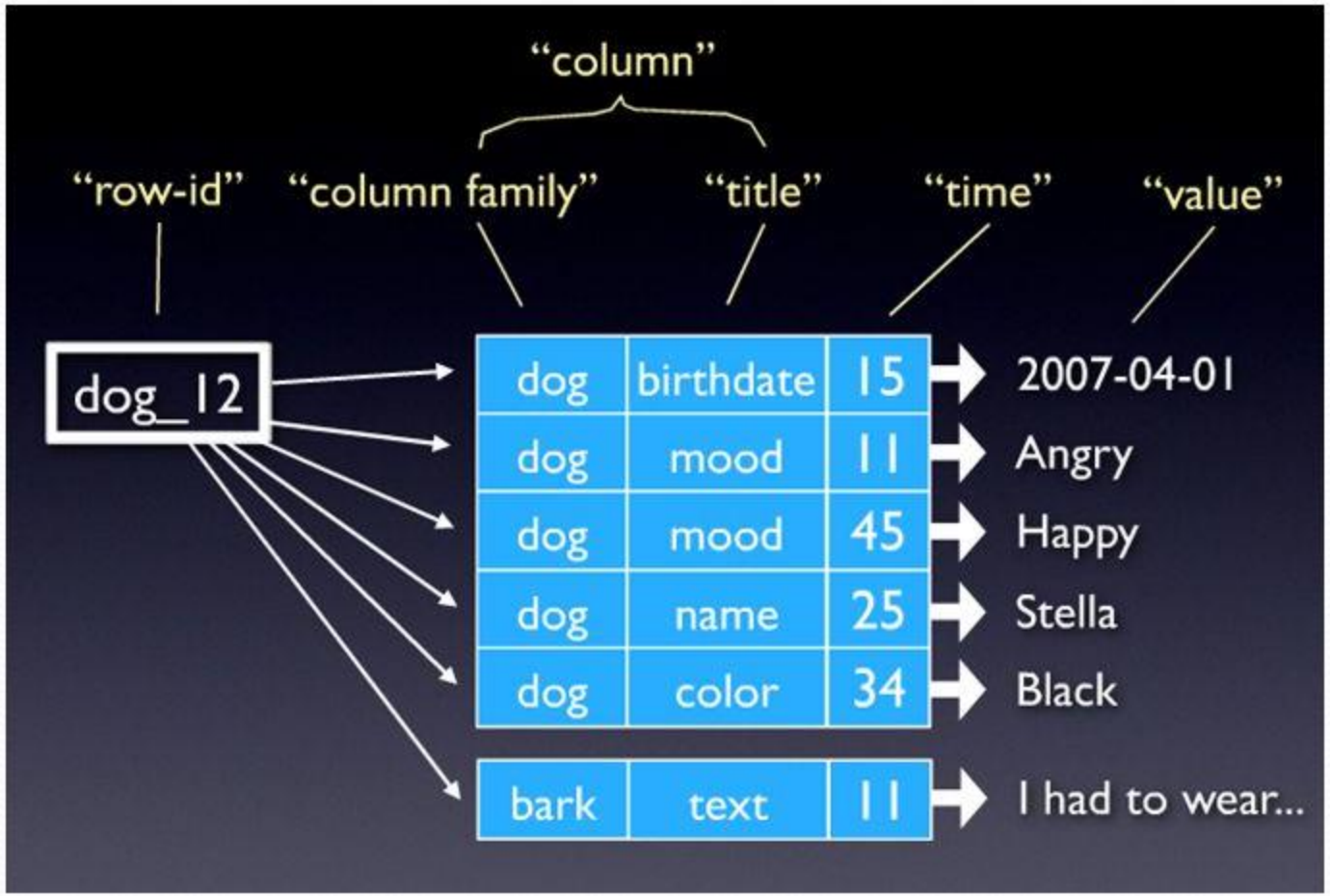
Graph databases

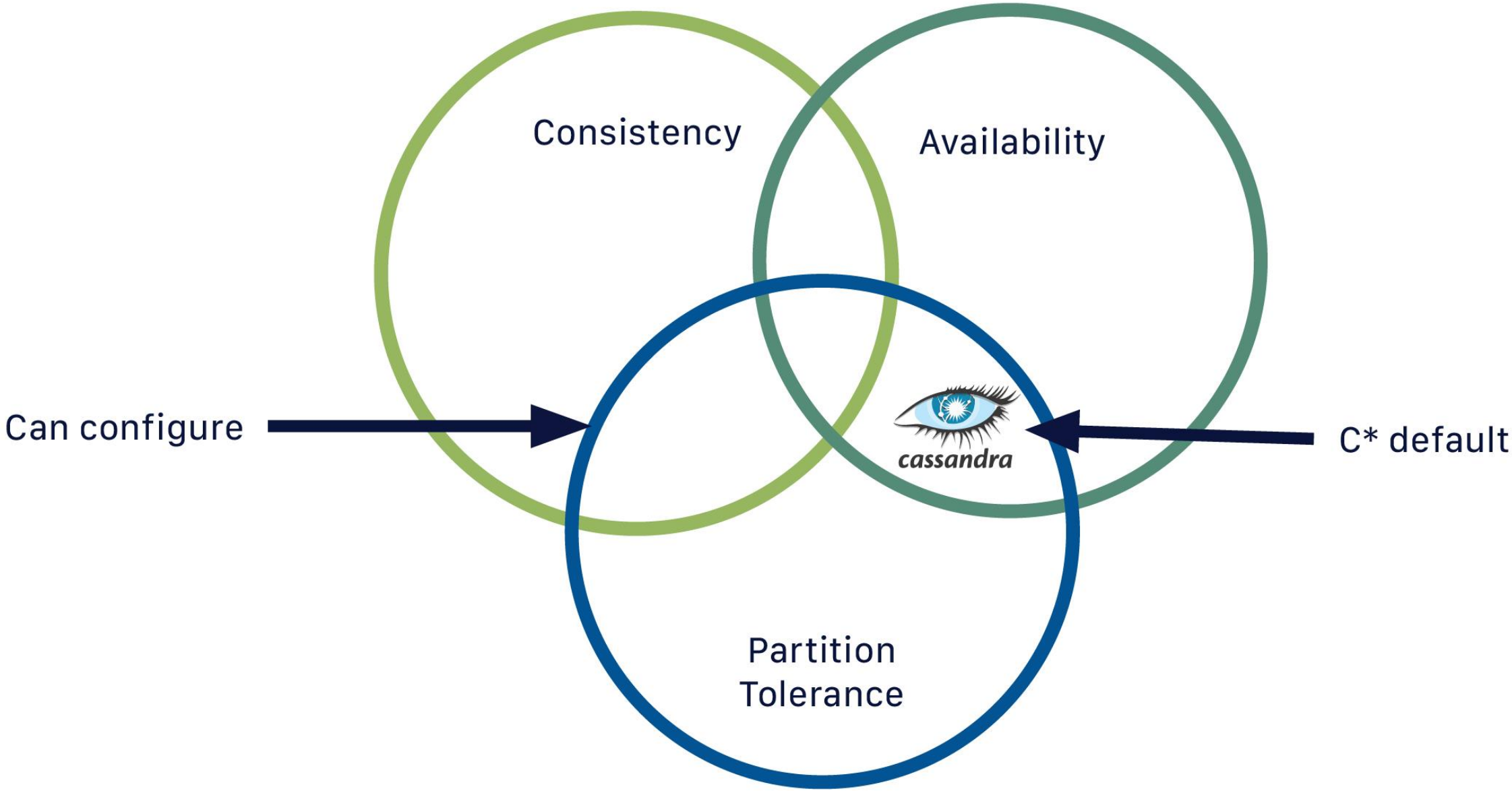


BIG TABLE-COLUMN IMPLEMENTATION

- In this Model, a key identifies a row, which contains data stored in one or more Column Families.
- Within a Column Family, each row can contain multiple columns.
- The values within each column are timestamped, so that several versions of a row-column mapping can live within a Column Family.
- The model naturally supports sparse column placement.
- It is particularly good at modeling historical data with timestamps.
- Main applications based on this store:
  - Cassandra
  - HBase
  - Hyper Table

Bigtable clones




Consistency
Availability
Partition
Tolerance
Can configure
C* default
cassandra

ADVANTAGES OF NOSQL

- Cheap & easy to implement due to open source.
- Easy to distribute
- Scale to available memory
- Have individual query language rather than using a standard query language
- Flexible data model
- ACID support is not required for product listing, status update etc
- Developer friendly-More developer centric interface

APPLICATIONS OF NOSQL

- Cassandra-: It is developed at Facebook & written in java. It uses Column Oriented & Eventual Consistency model.
- MongoDB-: It is an open source , high-performance, schema-free, document-oriented database written in the C++ programming language. It manages collections of JSON-like documents.
- In Yahoo ,Google or Amazon, you have had your data served via a NoSQL solution.
- In eBay or Twitter, you have indirectly used NOSQL datastores.

RDBMS vs. NoSQL

- Strong consistency vs. Eventual consistency
- Big dataset vs. HUGE datasets
- Scaling is possible vs. Scaling is easy
- SQL vs. Map-Reduce
- Good availability vs. Very high availability

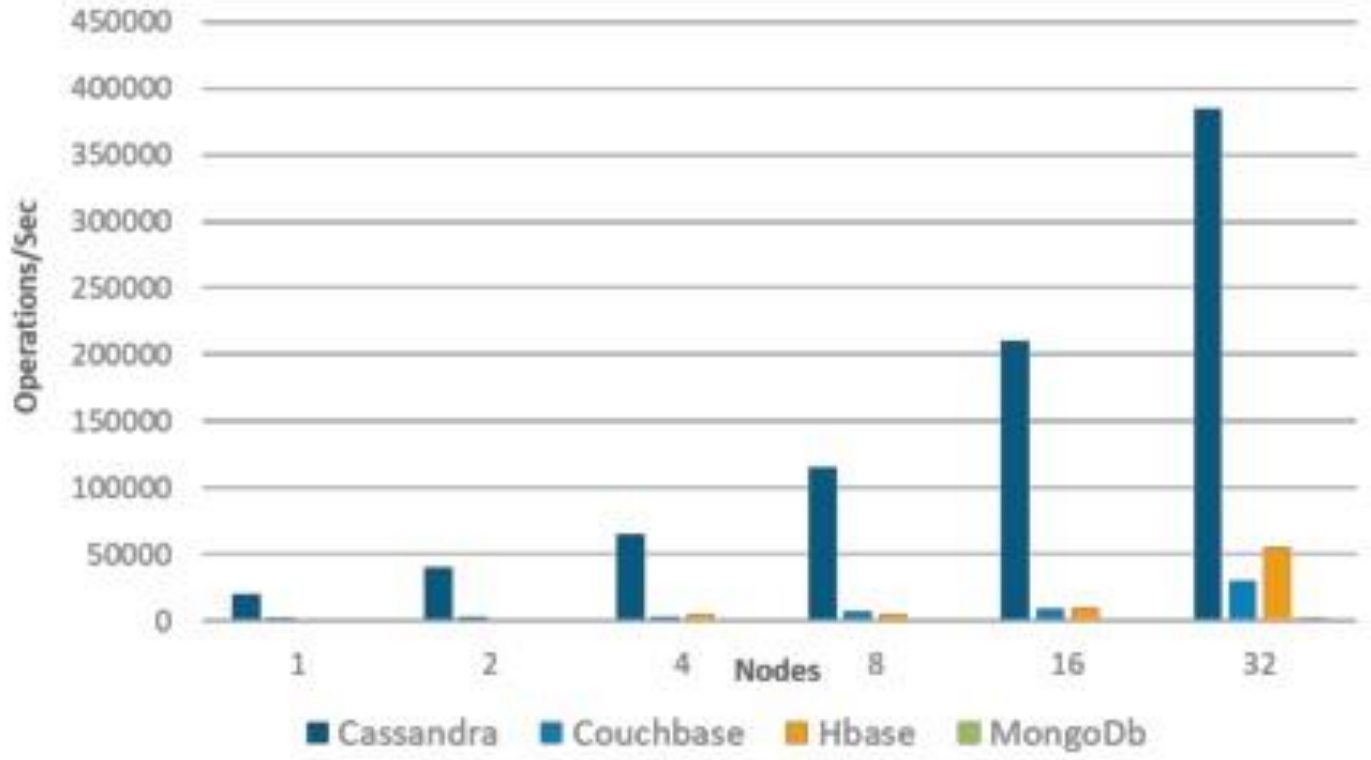
ACID

- Atomicity
  - All or Nothing
- Consistency
  - Valid according to all defined rules
- Isolation
  - No transaction should be able to interfere with another transaction
- Durability
  - Once a transaction has been committed, it will remain so, even in the event of power loss, crashes, or errors

BASE

- Basically Available
  - High availability but not always consistent
- Soft state
  - Background cleanup mechanism
- Eventual consistency
  - Given a sufficiently long period of time over which no changes are sent, all updates can be expected to propagate eventually through the system and all the replicas will be consistent.

Performance Comparison

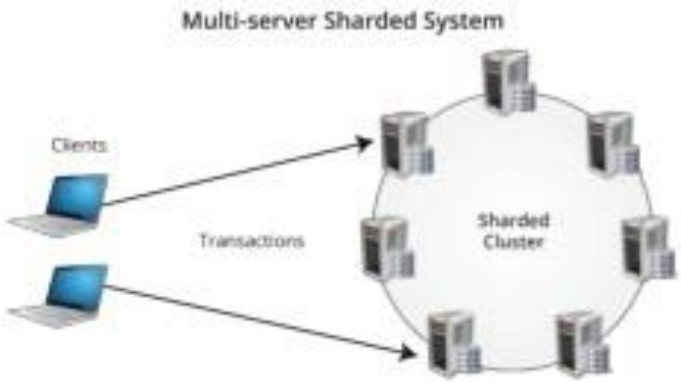


Requirements of today's application

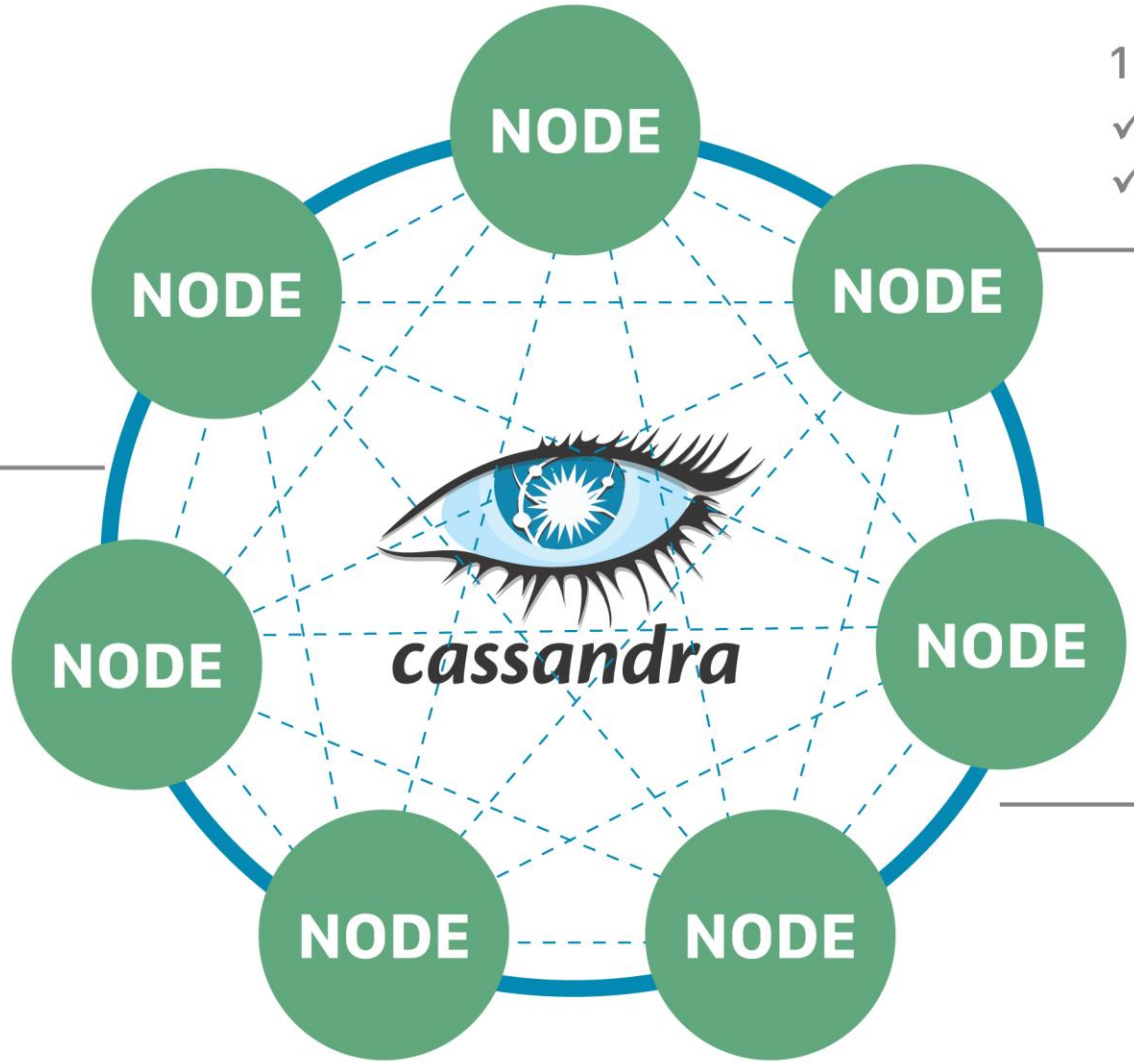
- Largescale, both in data footprint and query volume,
- Storage requirements
- Full global replication
- Always available
- Low-latency reads and writes

What is Apache Cassandra?

Apache Cassandra is a ***free and open-source distributed NoSQL database management system*** designed to ***handle large amounts of data*** across many commodity servers, providing ***high availability*** with no single point of failure.



ApacheCassandra™= NoSQL Distributed Database



1 Installation = 1 NODE
✓ Capacity + 1TB
✓ Throughput: 3000 Tx/sec/core

DataCenter | Ring

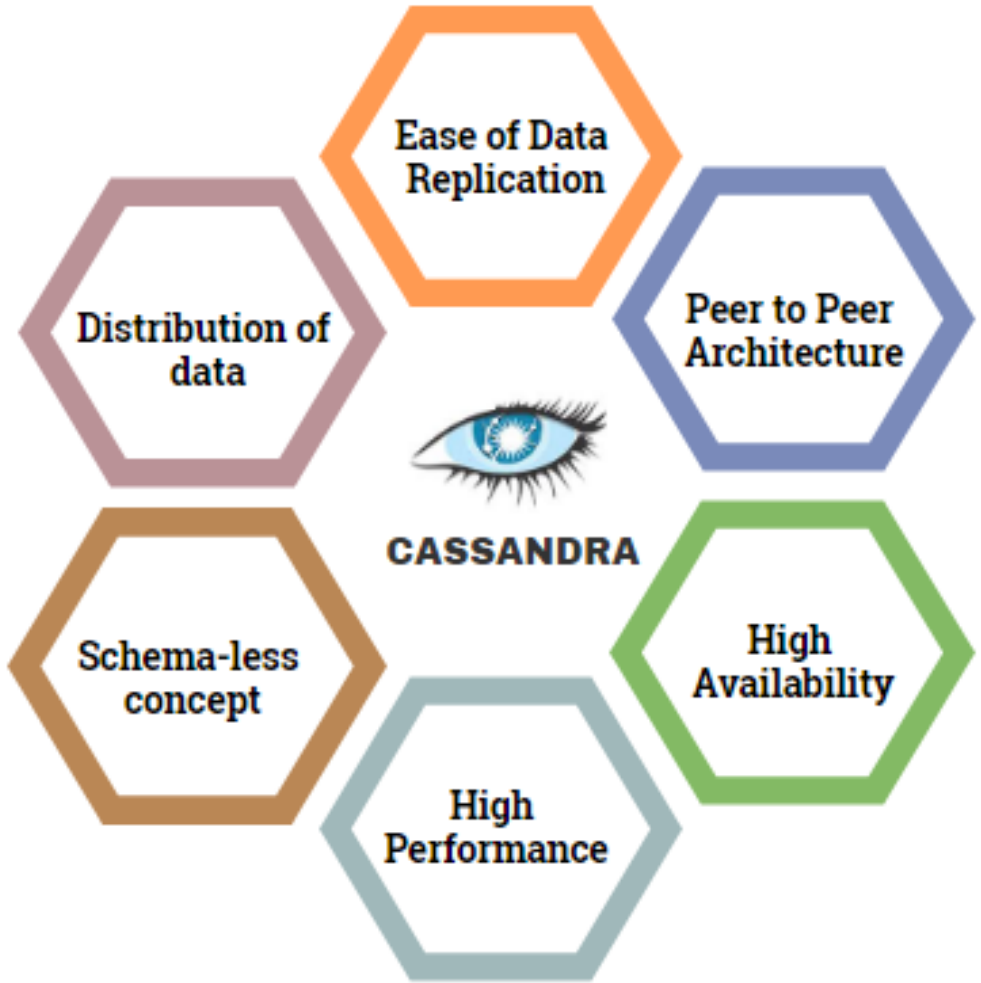
Communication:
✓ Gossiping

What is Apache Cassandra?

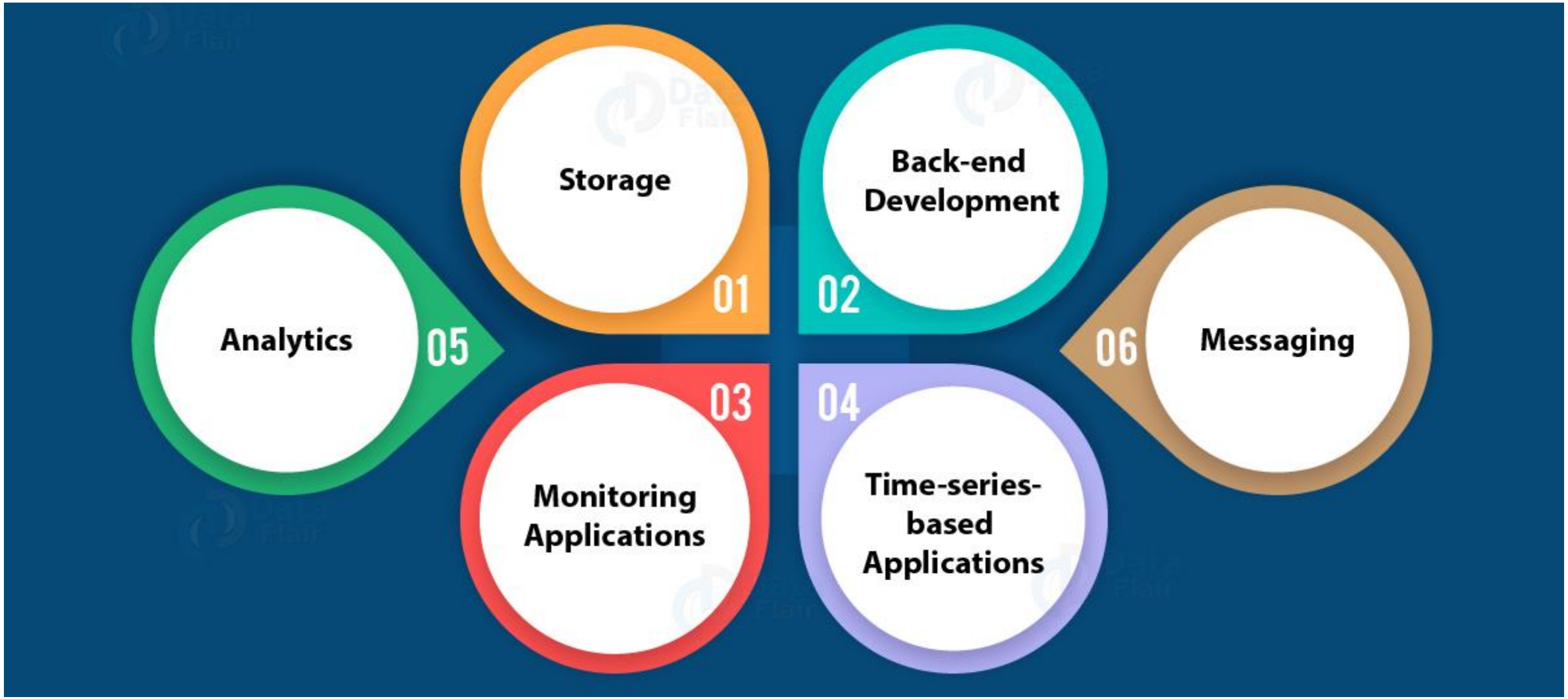
- Apache Cassandra is an open source NoSQL distributed database trusted by thousands of companies for scalability and high availability without compromising performance. Linear scalability and proven fault-tolerance on commodity hardware or cloud infrastructure make it the perfect platform for mission-critical data.

  -- Official Documentation

What is Cassandra

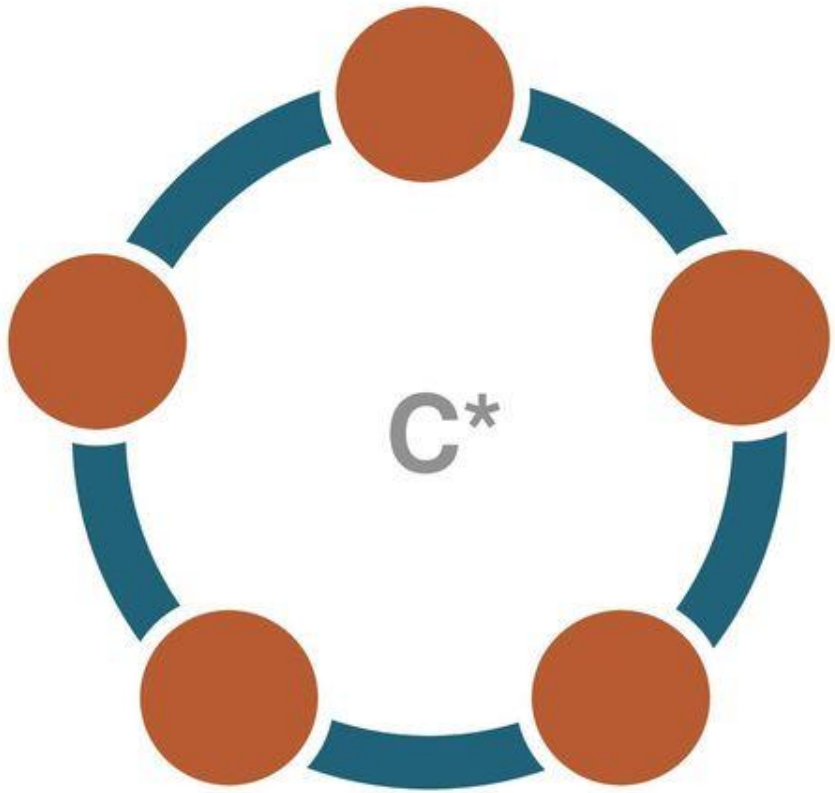


APPLICATIONS



Distributed Database

- ✓ Individual DBs (nodes)
- ✓ Working in a cluster
- ✓ Nothing is shared

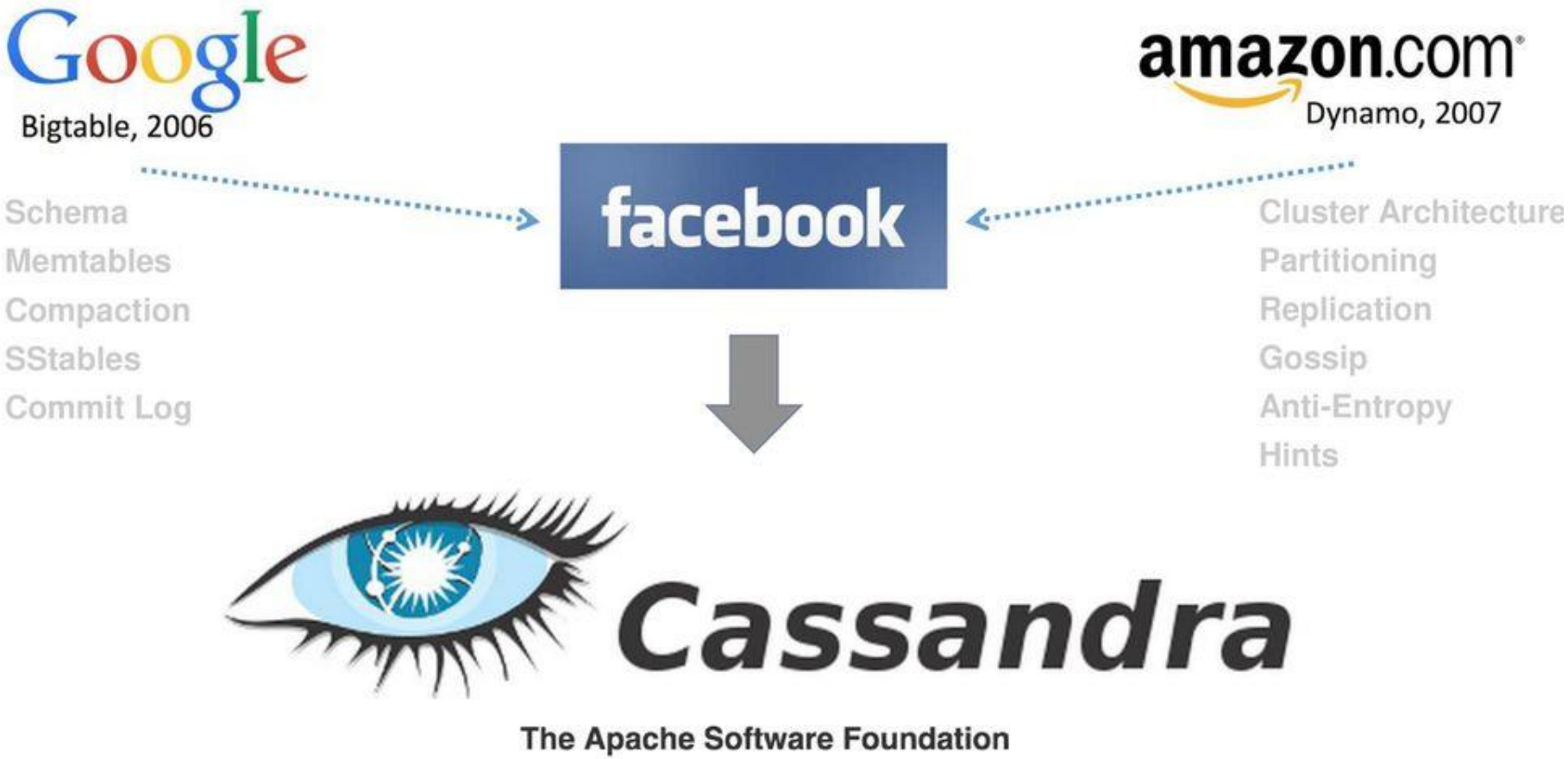


Cassandra according to CAP

- High availability is a priority in web based applications
- Cassandra chooses Availability and Partition Tolerance from the CAP guarantees, compromising on data Consistency to some extent.

Cassandra makes the following guarantees.

- High Scalability
- High Availability
- Durability
- Eventual Consistency of writes to a single table
- Lightweight transactions with linearizable consistency
- Batched writes across multiple tables are guaranteed to succeed completely or not at all
- Secondary indexes are guaranteed to be consistent with their local replicas data


Google
Bigtable, 2006
amazon.com
Dynamo, 2007
facebook
Schema
Memtables
Compaction
SStables
Commit Log
Cluster Architecture
Partitioning
Replication
Gossip
Anti-Entropy
Hints
Cassandra
The Apache Software Foundation

Google

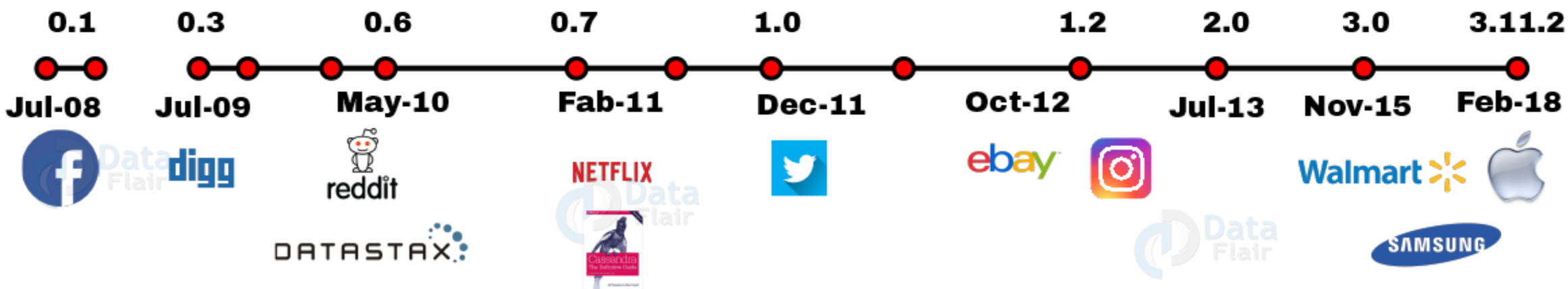
Bigtable, 2006

amazon.com

Dynamo, 2007

facebook

OpenSource, 2008



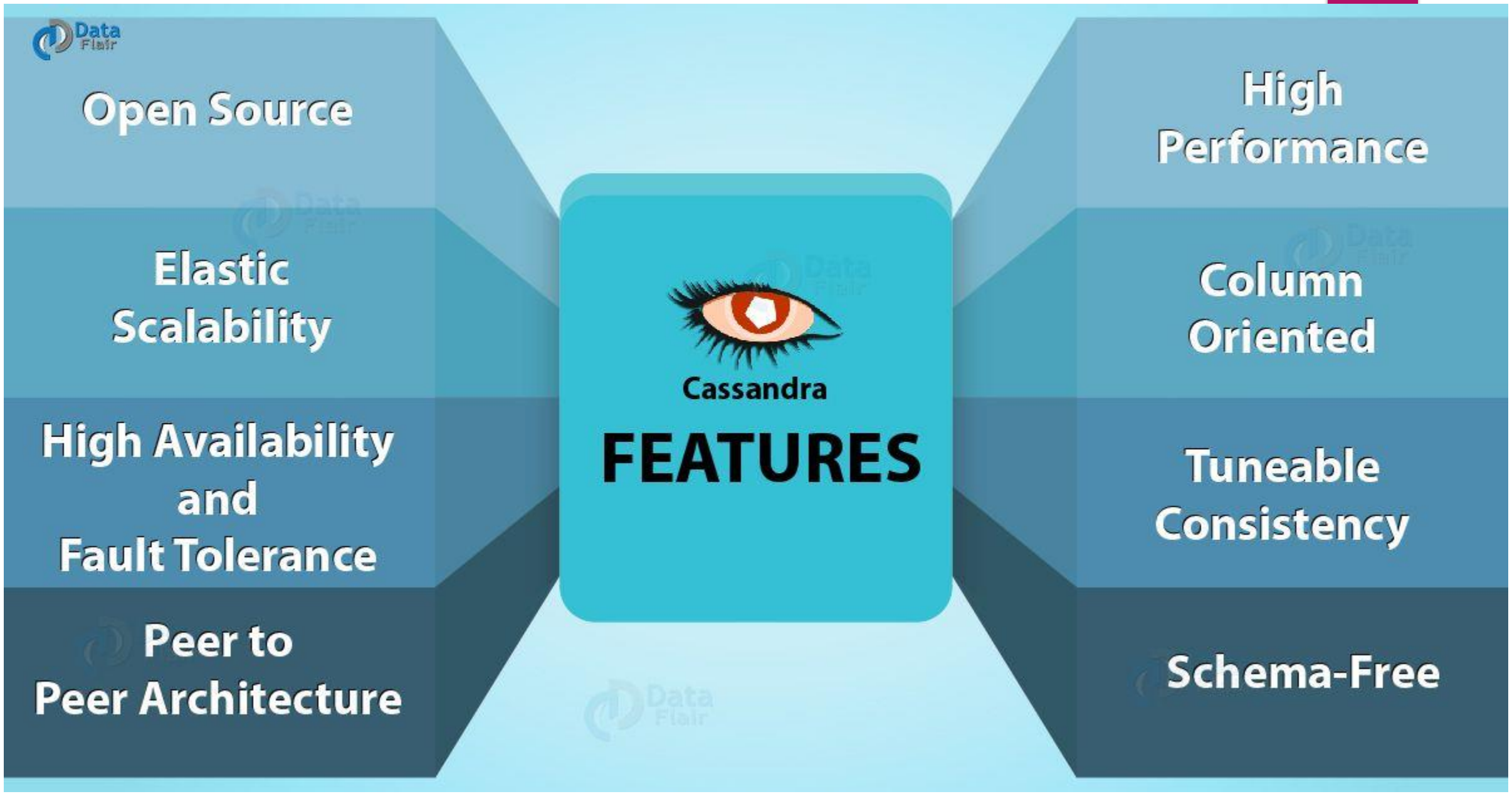
History of Cassandra

Cassandra vs RDBMS

Property	Cassandra	RDBMS
Core Architecture	Masterless (no single point of failure)	Master-slave (single points of failure)
High Availability	Always-on continuous availability	General replication with master-slave
Data Model	Dynamic; structured and unstructured data	Legacy RDBMS; Structured data
Scalability Model	Big data/Linear scale performance	Oracle RAC or Exadata
Multi-Data Center Support	Multi-directional, multi-cloud availability	Nothing specific
Enterprise Search	Integrated search on Cassandra data.	Handled via Oracle search
In-Memory Database Option	Built-in in-memory option	Columnar in-memory option

Design Objectives of Cassandra

- Full multi-master database replication
- Global availability at low latency
- Scaling out on commodity hardware
- Linear throughput increase with each additional processor
- Online load balancing and cluster growth
- Partitioned key-oriented queries
- Flexible schema


Data Flair
Open Source
Elastic Scalability
High Availability and Fault Tolerance
Peer to Peer Architecture
Cassandra
FEATURES
High Performance
Column Oriented
Tuneable Consistency
Schema-Free

Features of Cassandra

Features of Cassandra

- Distributed and Decentralized
- Elastic Scalability
- High Availability and Fault Tolerance
- Tuneable Consistency
- Row-Oriented
- Schema-Free
- High Performance

Cassandra

- Open source, distributed NoSQL database that began internally at Facebook and was released as an open-source project in July 2008.
- Delivers continuous availability (zero downtime), high performance, and linear scalability that modern applications require
- Offers operational simplicity and effortless replication across data centers and geographies.
- Can handle petabytes of information and thousands of concurrent operations per second, enabling organizations to manage large amounts of data across hybrid cloud and multi cloud environments.

Open Source

- Cassandra, though it is very powerful and reliable, is FREE!
- An open source project by Apache.
  - Huge Cassandra Community, where people discuss their queries and views.
- Possibility of integrating Cassandra with other Apache Open-source projects like Hadoop, Apache Pig, Apache Hive etc
- Are attractive because of their affordability and extensibility, as well as the flexibility to avoid vendor lock-in.
- Organizations adopting open source report higher speed of innovation and faster adoption.

Peer-to-peer Architecture

- Some databases work on master-slave architecture and some work on peer-to-peer
- In master-slave architecture, there is the main unit and the rest communicate with that unit.
- Whereas, in a peer-to-peer architecture, several units communicate with each other.
- Apache Cassandra follows peer-to-peer architecture.
- There is no single point of failure.
- Cassandra has a robust architecture with exceptional characteristics.
- In Cassandra, no single node is in charge of replicating data across a cluster
- Every node is capable of performing all read and write operations. This improves performance and adds resiliency to the database.

Elastic Scalability

- Can easily scale-up or scale-down the cluster in Cassandra.
- Flexibility for adding or deleting any number of nodes from the cluster without disturbances.
- No need of restarting the cluster while scaling up or scaling down. Because of this, Cassandra has a very high throughput for the highest number of nodes.
- Zero downtime or any pause during scaling. Hence read and write throughput increases simultaneously without delay.

High Availability and Fault Tolerance

- Data Replication is the storage of data at multiple locations
- Because of data replication, Cassandra is highly available and fault tolerant.
- Basically, if one node fails, the data is readily available in different nodes.
- Therefore, can retrieve the data from those nodes.
- User sets the number of replication.
- According to that number, you can replicate each row in a cluster based on the row key.
- Data replication can be across multiple data centers. Evidently, this leads to high-level back-up and recovery competencies.

Active Everywhere / Zero-Downtime

- Since every Cassandra node is capable of performing read and write operations, data is quickly replicated across hybrid cloud environments and geographies.
- In the event a node fails, users are automatically routed to the nearest healthy node.
- They won't even notice that a node has been knocked offline because applications behave as designed even in the event of failure.
- As a result, applications are always available and data is always accessible and never lost.
- Cassandra's built-in repair services fix problems immediately after they occur—without any manual intervention.
- Productivity doesn't even need to take a hit should nodes fail.


Adjustable Replication
Work Load Segregation
High Availability
Multiple Data Centre
Ring Structure

High Performance

- Cassandra database has one of the best performance as compared to other NoSQL database.
- Developers wanted to utilize the capabilities of many multi-core machines. This is the base of development of Cassandra.
- Cassandra has proven itself to be excellently reliable when it comes to a large set of data.
- Cassandra is used by a lot of organizations which deal with huge amount of data on a daily basis. They are ensured about the data, as they cannot afford to lose the data.
- Cassandra consistently outperforms popular NoSQL alternatives in benchmarks and real applications, primarily because of fundamental architectural choices.

ColumnOriented

- Cassandra's data model is column-oriented.
- In other databases, column name contains metadata, whereas, columns in Cassandra also contain actual data.
- In Cassandra, columns are stored based on column names.
- Therefore, there are a number of columns contained in rows.

Tunable Consistency

- Basically, there are two types of consistency in Cassandra, Eventual consistency and Strong Consistency.
- The developer can choose any of them according to his requirement.
- As soon as the cluster accepts the write, eventual consistency makes it sure that the client approves.
- Whereas, Strong consistency make sure that any update is broadcasted to all the nodes or machines where the particular data is suited.
- Mixture of the two consistency is also a possibility.

Schema-Free

- There is a flexibility in Cassandra to create columns within the rows.
- Cassandra is known as the schema-optional data model.
- Since each row may not have the same set of columns, there is no need to show all the columns needed by the application at the surface.
- Therefore, Schema-less/Schema-free database in a column family is one of the most important Cassandra features.

Distributed and Decentralized

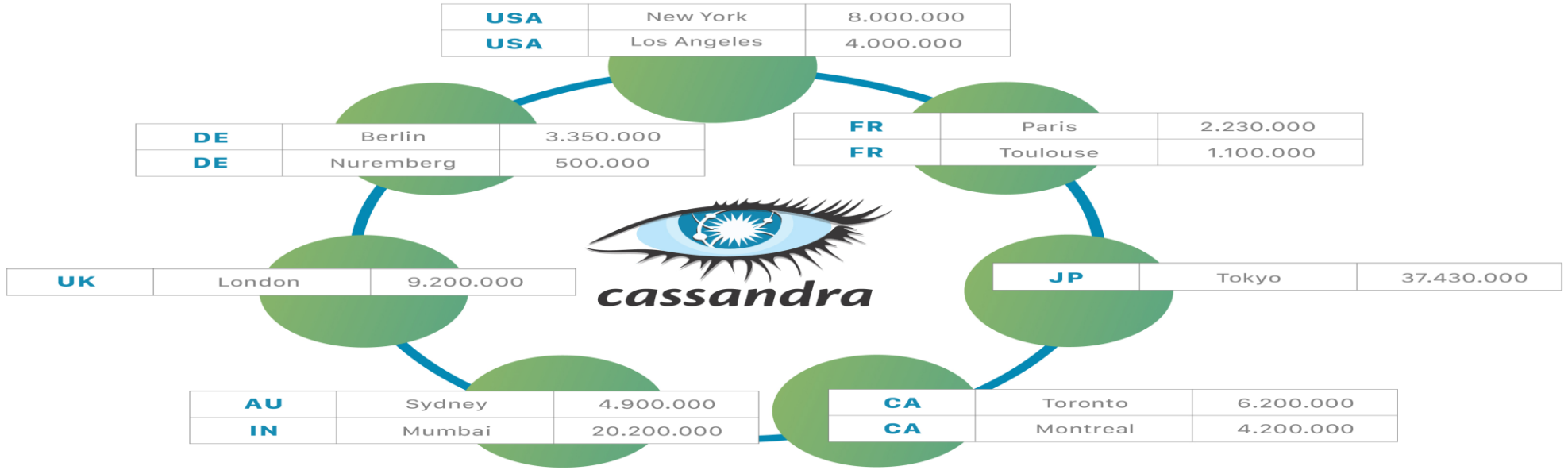
- databases are distributed.
- Yields both technical and business advantages.
- Cassandra databases easily scale when an application is under high stress, and the distribution also prevents data loss from any given datacenter's hardware failure.
- A distributed architecture also brings technical power; for example, a developer can tweak the throughput of read queries or write queries in isolation.
- "Distributed" means that Cassandra can run on multiple machines while appearing to users as a unified whole

Distributed and Decentralized

- Cassandra can (and usually does) have multiple nodes.
- A node represents a single instance of Cassandra.
- These nodes communicate with one another through a protocol called gossip, which is a process of computer peer-to-peer communication.
- Cassandra also has a masterless architecture – any node in the database can provide the exact same functionality as any other node – contributing to Cassandra's robustness and resilience.
- Multiple nodes can be organized logically into a cluster, or "ring".
- Can also have multiple datacenters.

COUNTRY	CITY	POPULATION
USA	New York	8.000.000
USA	Los Angeles	4.000.000
FR	Paris	2.230.000
DE	Berlin	3.350.000
UK	London	9.200.000
AU	Sydney	4.900.000
DE	Nuremberg	500.000
CA	Toronto	6.200.000
CA	Montreal	4.200.000
FR	Toulouse	1.100.000
JP	Tokyo	37.430.000
IN	Mumbai	20.200.000

Partition Key



Elastic Scalability

- Enables developers to scale their databases dynamically, using off-the-shelf hardware, with no downtime.
- Can expand when you need to – and also shrink, if the application requirements suggest that path.
- Cassandra makes it easy to increase the amount of data it can manage.
- Because it's based on nodes, Cassandra scales horizontally (aka scale-out), using lower commodity hardware.
- To double your capacity or double your throughput, double the number of nodes. That's all it takes.
- Need more power? Add more nodes – whether that's 8 more or 8,000 – with no downtime.
- Also have the flexibility to scale back if you wish.
- This linear scalability applies essentially indefinitely. This capability has become one of Cassandra's key strengths.

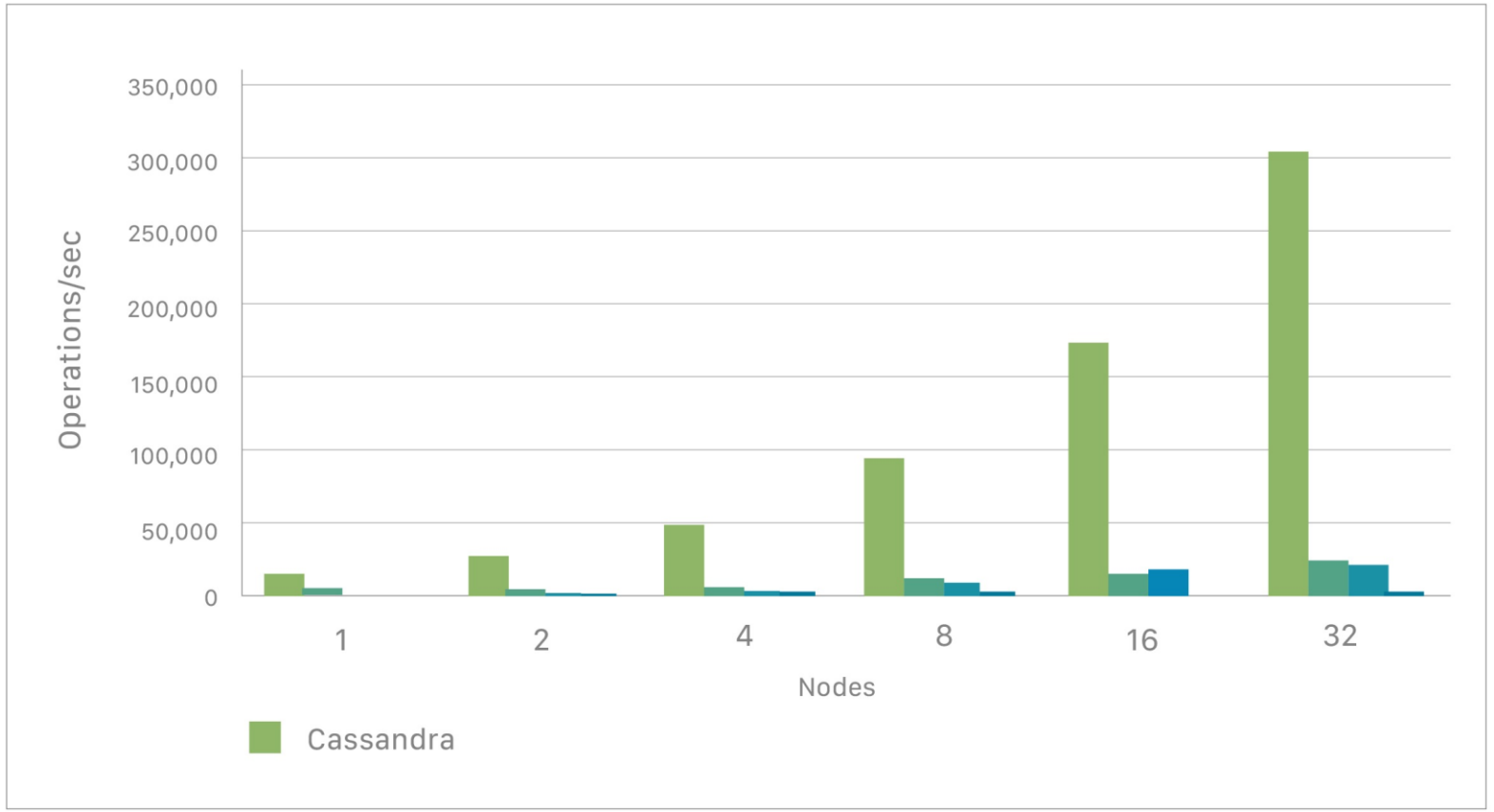
Scales Linearly

Need more capacity?

Need more throughput?

Add nodes!

Balanced Read/Write Mix



COUNTRY	CITY	POPULATION
AU	Sydney	4.900.000
CA	Toronto	6.200.000
CA	Montreal	4.200.000
DE	Berlin	3.350.000
DE	Nuremberg	500.000

Partition Key

Partitioner

Hashing Function

COUNTRY	CITY	POPULATION
59	Sydney	4.900.000
12	Toronto	6.200.000
12	Montreal	4.200.000
45	Berlin	3.350.000
45	Nuremberg	500.000

Tokens

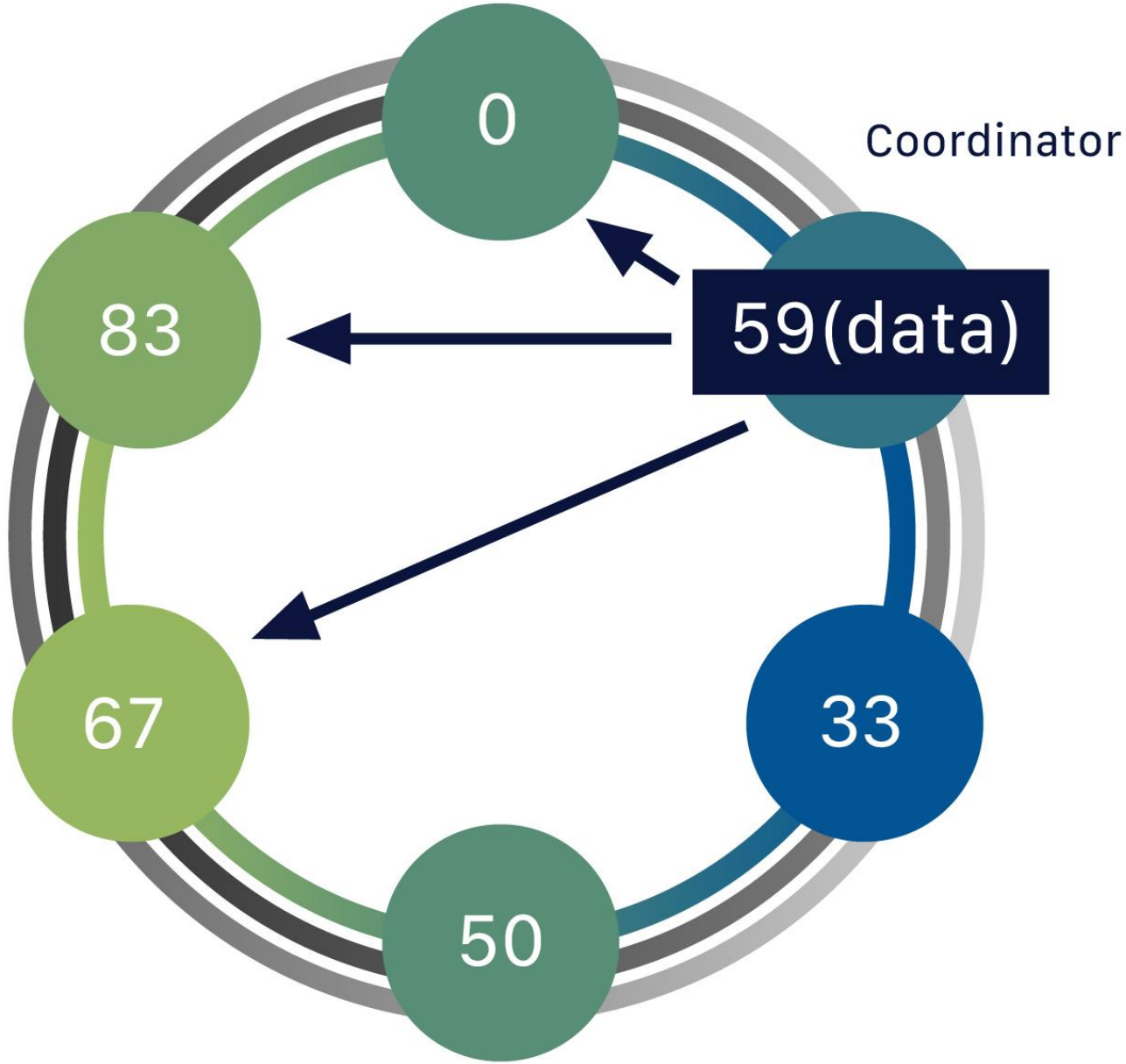
Partitions in cassandra

- In Cassandra, the data itself is automatically distributed, with (positive) performance consequences.
- Accomplishes this using partitions.
- Each node owns a particular set of tokens, and Cassandra distributes data based on the ranges of these tokens across the cluster.
- Partition key is responsible for distributing data among nodes and is important for determining data locality.
- When data is inserted into the cluster, the first step is to apply a hash function to the partition key.
- The output is used to determine what node (based on the token range) will get the data.

Partitions in cassandra

- When data comes in, the database's coordinator takes on the job of assigning to a given partition – let's call it partition 59.
- Any node in the cluster can take on the role as the coordinator.
- nodes gossip to one another; during which they communicate about which node is responsible for what ranges.
- So in our example, the coordinator does a lookup: Which node has the token 59? When it finds the right one, it forwards that data to that node.
- The node that owns the data for that range is called a replica node.
- One piece of data can be replicated to multiple (replica) nodes, ensuring reliability and fault tolerance. So far, our data has only been replicated to one replica. This represents a replication factor of one, or RF = 1.
- The coordinator node isn't a single location; the system would be fragile if it were. It's simply the node that gets the request at that particular moment. Any node can act as the coordinator.

RF = 3



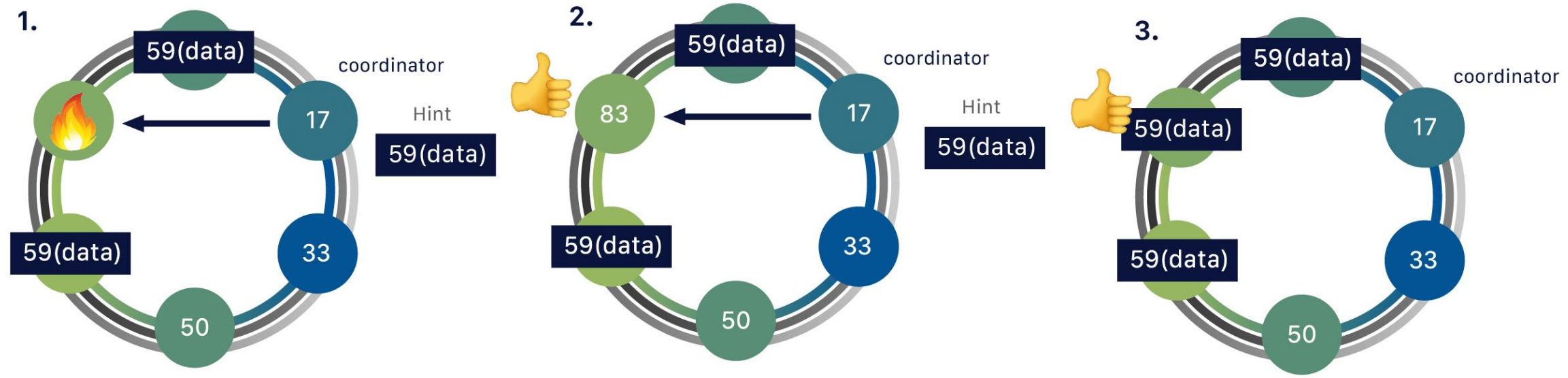
High Availability and Fault Tolerance

- One piece of data can be replicated to multiple (replica) nodes, ensuring reliability and fault tolerance.
- Cassandra supports the notion of a replication factor (RF), which describes how many copies of your data should exist in the database.
- For a replication factor of two (RF = 2), the data needs to be stored on a second replica as well – and hence each node becomes responsible for a secondary range of tokens, in addition to its primary range.
- A replication factor of three ensures that there are three nodes (replicas) covering that particular token range, and the data is stored on yet another one.

High Availability and Fault Tolerance

- If a node goes down, or a hard drive fails, or AWS resets an instance, Replication ensures that data isn't lost.
- If a request comes in for data, even if one of our replicas has gone down, the other two are still available to fulfill the request.
- The coordinator stores a "hint" for that data as well, and when the downed replica comes back up, it will find out what it missed, and catch up to speed with the other two replicas.
- No manual action is required, this is done completely automatically.

RF = 3

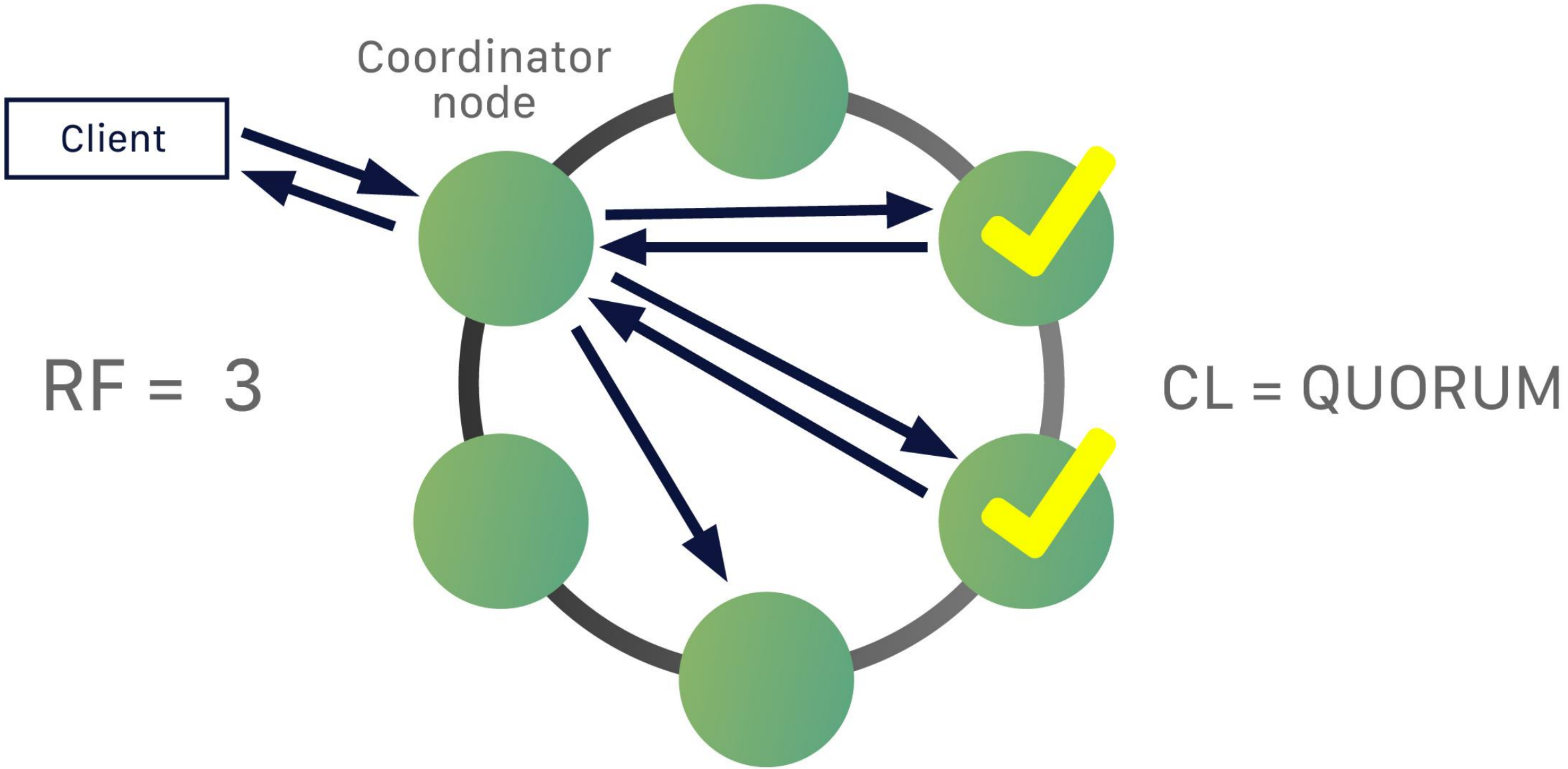

1.
59(data)
17
33
50
59(data)
coordinator
Hint
59(data)
2.
59(data)
83
17
33
50
59(data)
coordinator
Hint
59(data)
3.
59(data)
17
33
50
59(data)
59(data)
coordinator

Tuneable Consistency

- Cassandra is by default an AP (Available Partition-tolerant) database, hence it is “always on”.
- Can indeed configure the consistency on a per-query basis.
- Consistency level represents the minimum number of Cassandra nodes that must acknowledge a read or write operation to the coordinator before the operation is considered successful.
- As a general rule, you will select your consistency level (CL) based on your replication factor.

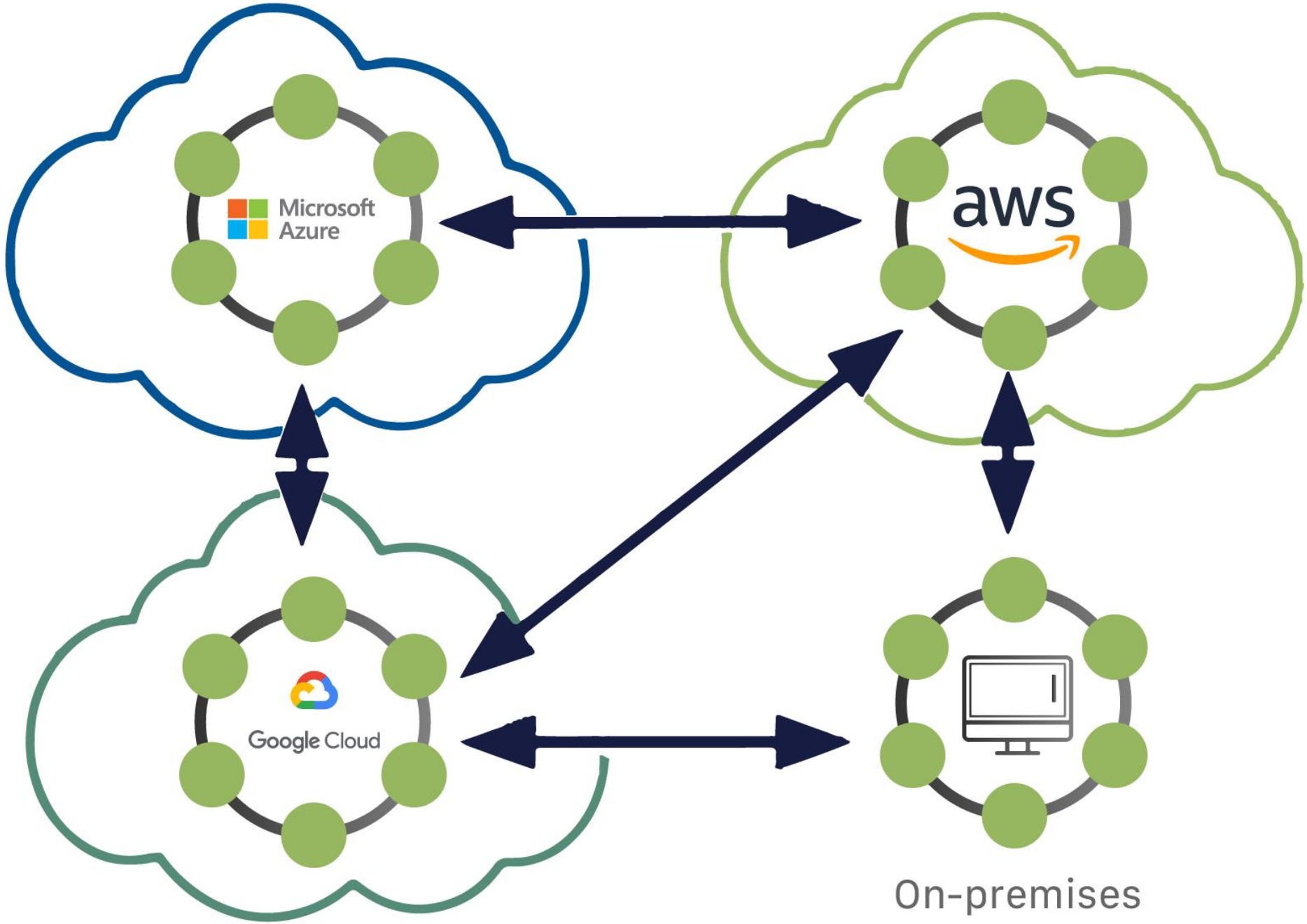
Tuneable Consistency

- If data is replicated out to three nodes.
- Can have a CL=QUORUM (Quorum referring to majority, 2 replicas in this case or RF/2 +1) therefore the coordinator will need to get acknowledgement back from two of the replicas in order for the query to be considered a success.
- As with other computing tasks, it can take some skill to learn to tune this feature for ideal performance, availability, and data integrity – but the fact that you can control it with such granularity means you can control deployments in great detail.


Client
Coordinator node
RF = 3
CL = QUORUM

deployment agnostic

- Cassandra is deployment agnostic
- Doesn't care where you put it – on prem, a cloud provider, multiple cloud providers.
- Can use a combination of those for a single database.
- That gives software developers the maximum amount of flexibility.


Microsoft Azure
aws
Google Cloud
On-premises

To sum up

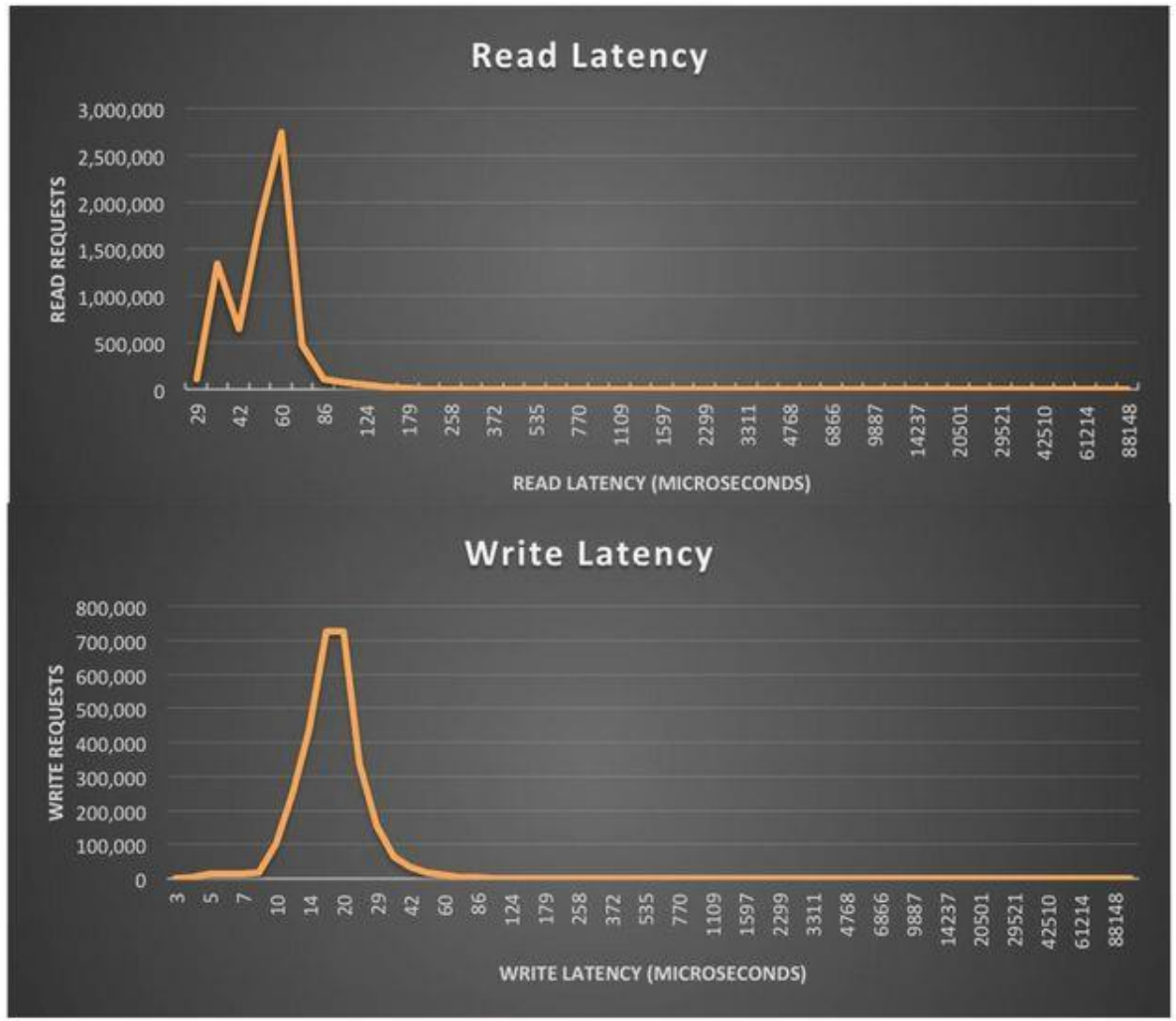
- Full multi-master database replication
- Global availability at low latency
- Scaling out on commodity hardware
- Linear throughput increase with each additional processor
- Online load balancing and cluster growth
- Partitioned key-oriented queries
- Flexible schema

Cassandra Use Case

Cassandra Use Case

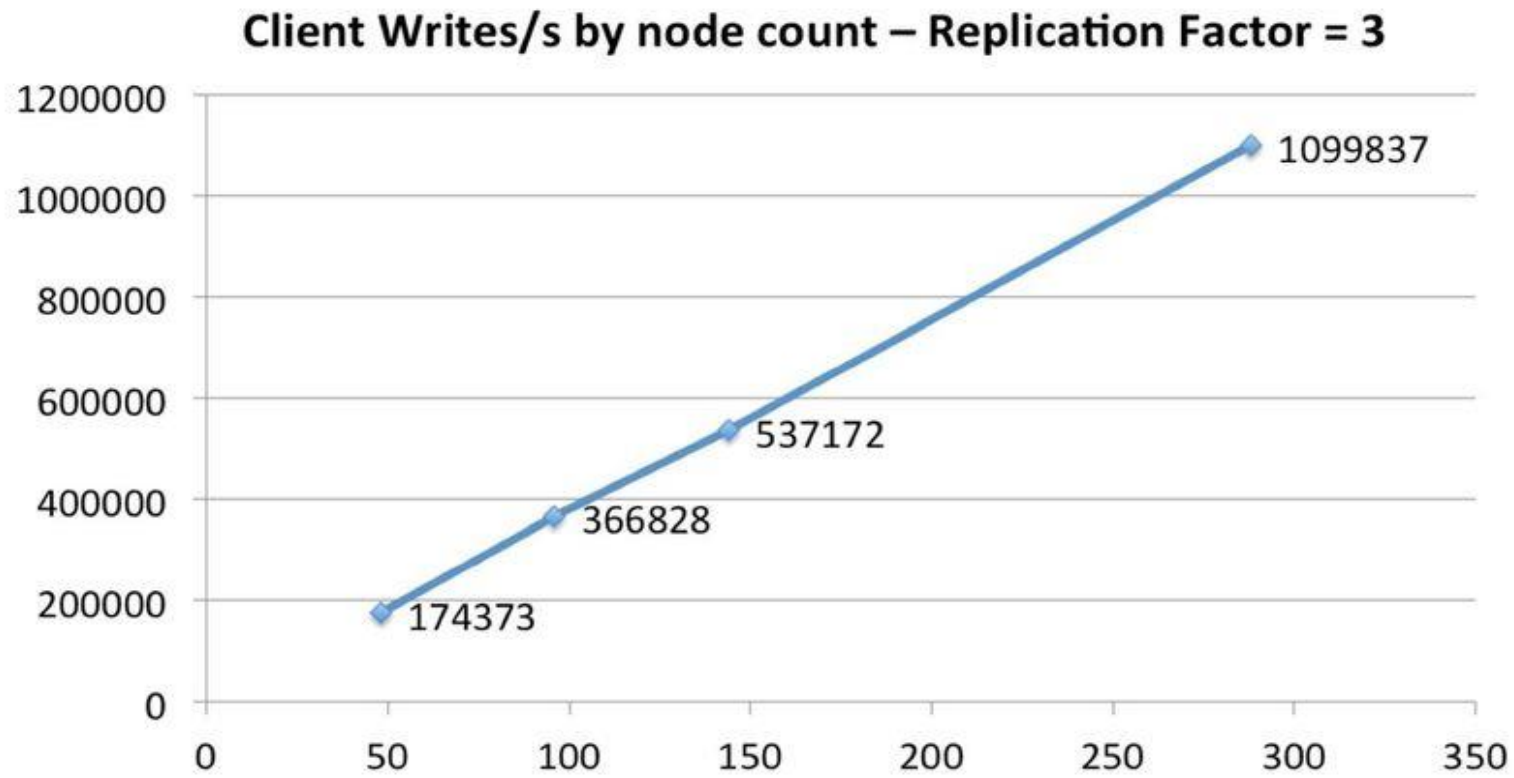
Why Cassandra?

It's Fast (Low Latency)



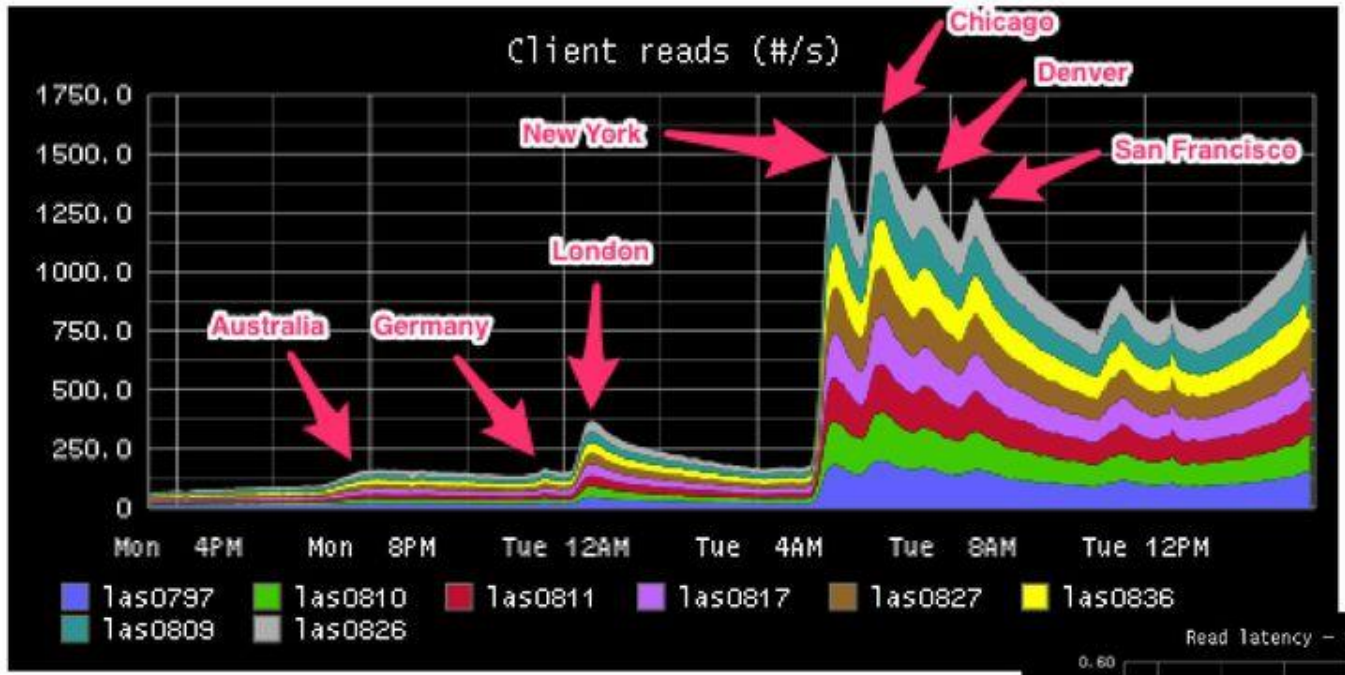
Why Cassandra?

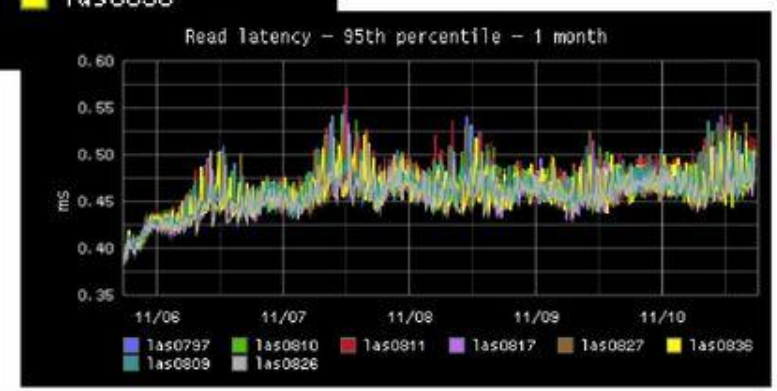
It's Hugely Scalable (High Throughput)



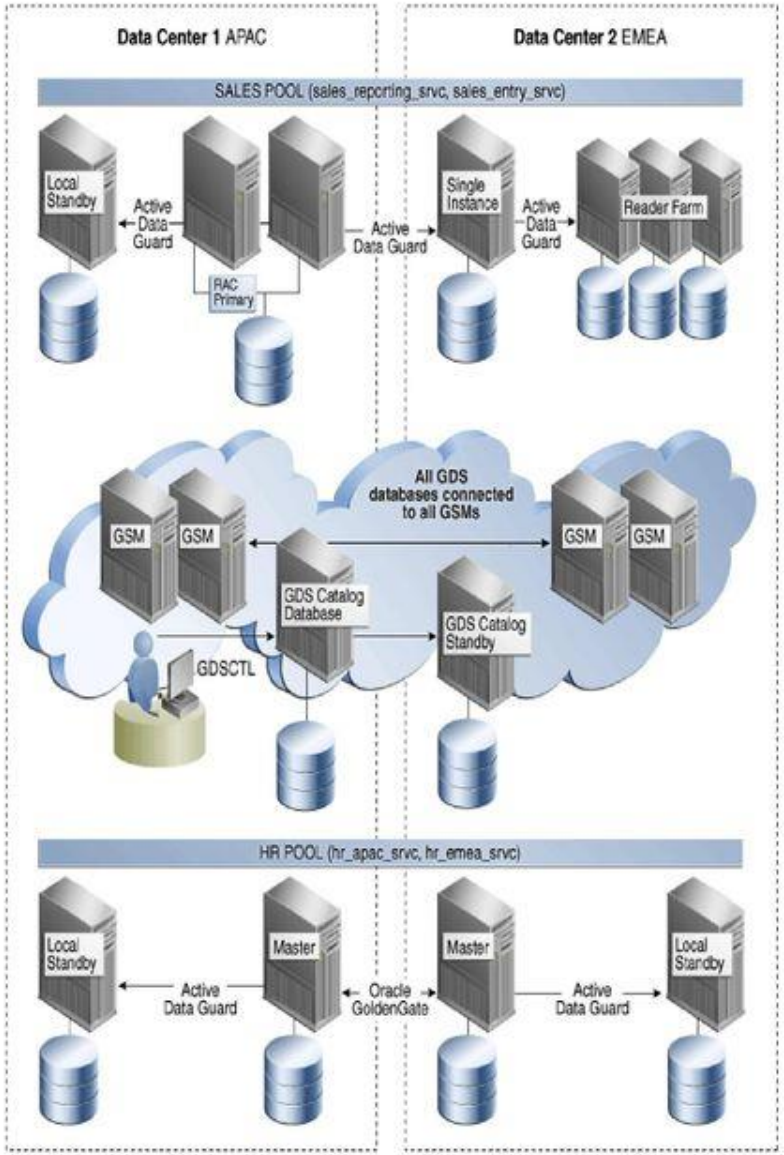
Why Cassandra?

It is natively multi-data center with distributed data

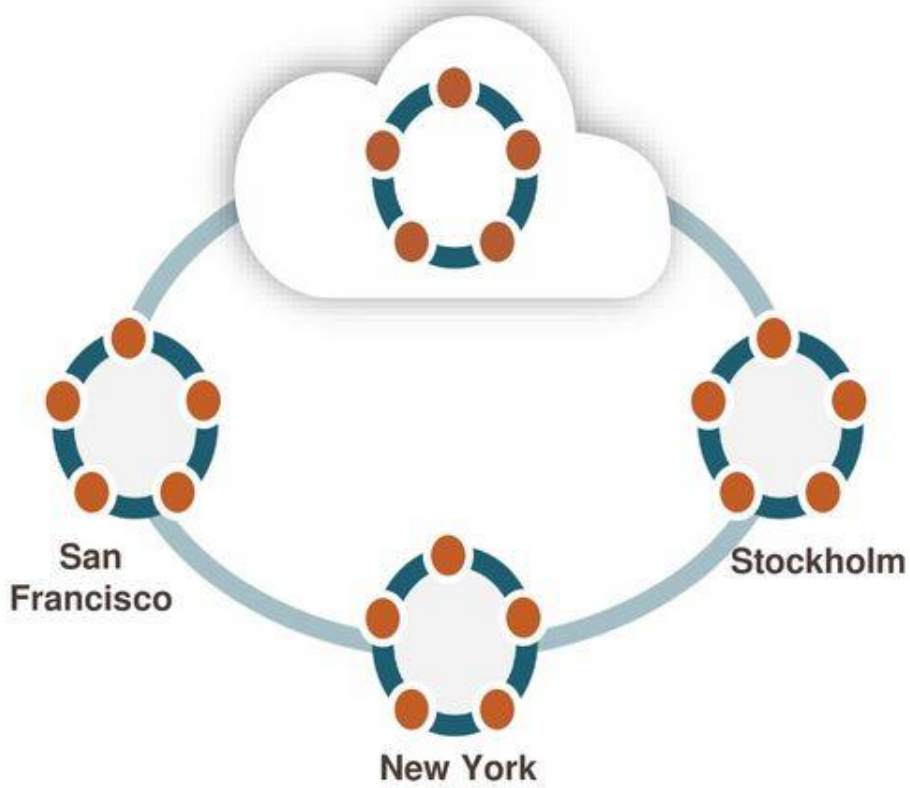




Operational Simplicity



OR



Cassandra Query Language (CQL)

- Cassandra Query Language (CQL), an SQL-like language, to create and update database schema and access data.
- CQL allows users to organize data within a cluster of Cassandra nodes using:
- **Keyspace**: defines how a dataset is replicated, for example in which datacenters and how many copies. Keyspaces contain tables.
- **Table**: defines the typed schema for a collection of partitions. Cassandra tables have flexible addition of new columns to tables with zero downtime. Tables contain partitions, which contain partitions, which contain columns.
- **Partition**: defines the mandatory part of the primary key all rows in Cassandra must have. All performant queries supply the partition key in the query.
- **Row**: contains a collection of columns identified by a unique primary key made up of the partition key and optionally additional clustering keys.
- **Column**: A single datum with a type which belong to a row.

Cassandra Query Language (CQL)

Supports numerous advanced features over a partitioned dataset such as:

- Single partition lightweight transactions with atomic compare and set semantics.
- User-defined types, functions and aggregates
- Collection types including sets, maps, and lists.
- Local secondary indices
- (Experimental) materialized views

Cassandra – Not suitable for

- Cassandra explicitly chooses not to implement operations that require cross partition coordination as they are typically slow and hard to provide highly available global semantics.

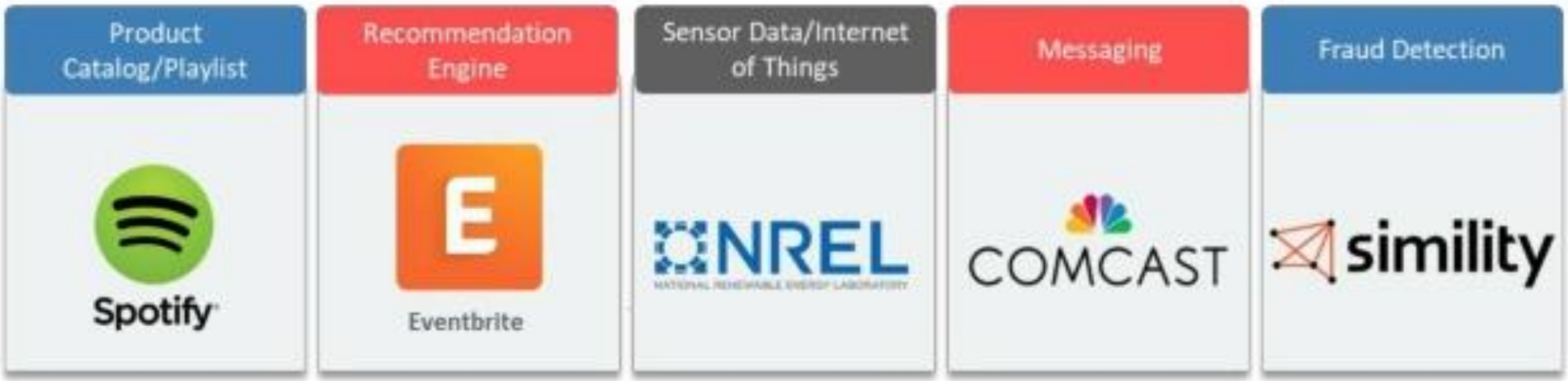
For example Cassandra does not support:

- Cross partition transactions
- Distributed joins
- Foreign keys or referential integrity.

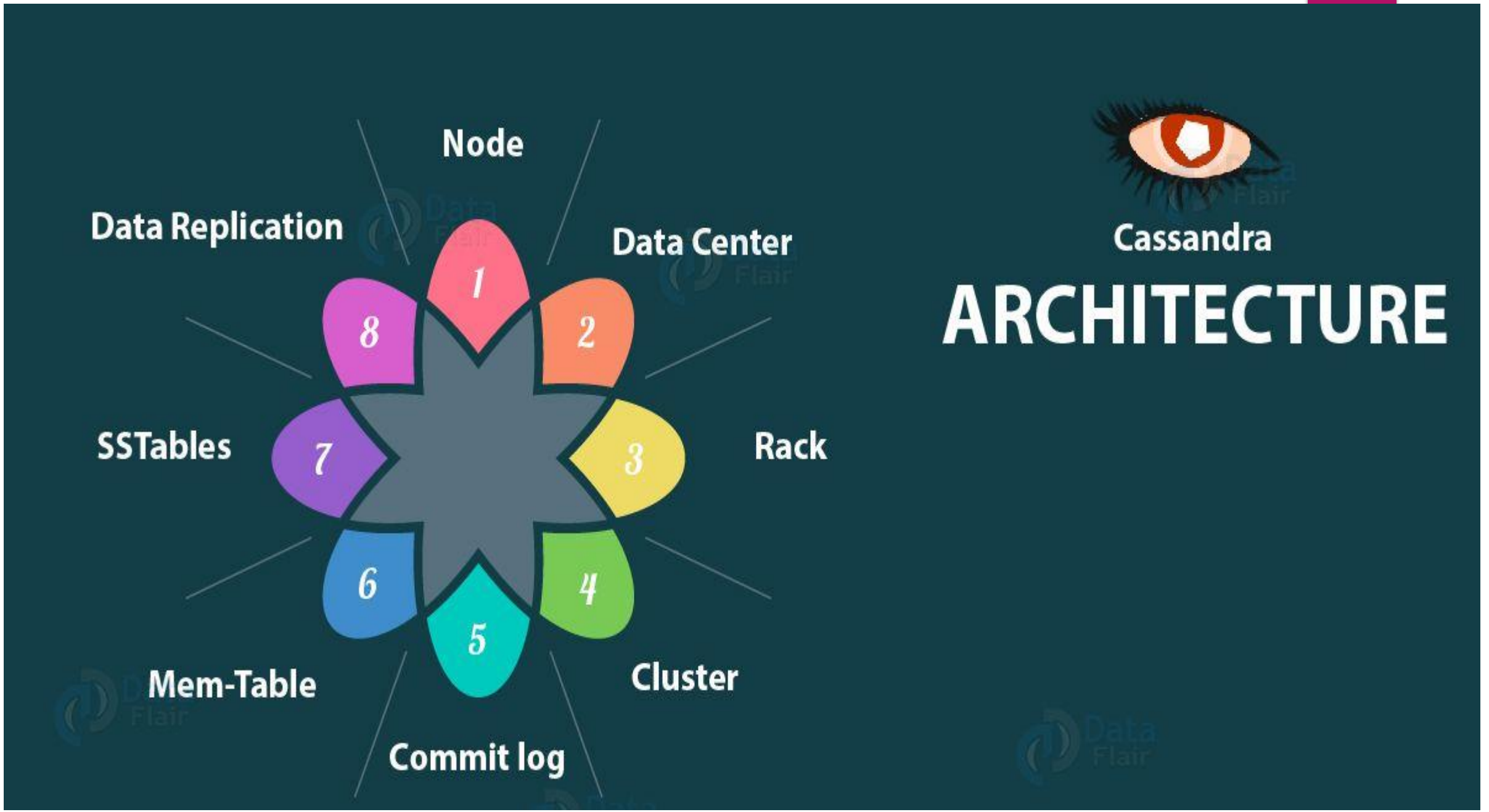
Tools with Cassandra

- **cassandra.yaml** - Contains configuration settings
  - Can be edited by hand or with the aid of configuration management tools.
  - Some settings can be manipulated live using an online interface, but others require a restart of the database to take effect.
- **nodetool** command - Interacts with Cassandra's live control interface, allowing runtime manipulation of many settings from cassandra.yaml.
- auditlogviewer - Used to view the audit logs.
- fqltool - View, replay and compare full query logs.
- Supports out of the box atomic snapshot functionality, which presents a point in time snapshot of Cassandra's data for easy integration with many backup tools.
- Supports incremental backups where data can be backed up as it is written.

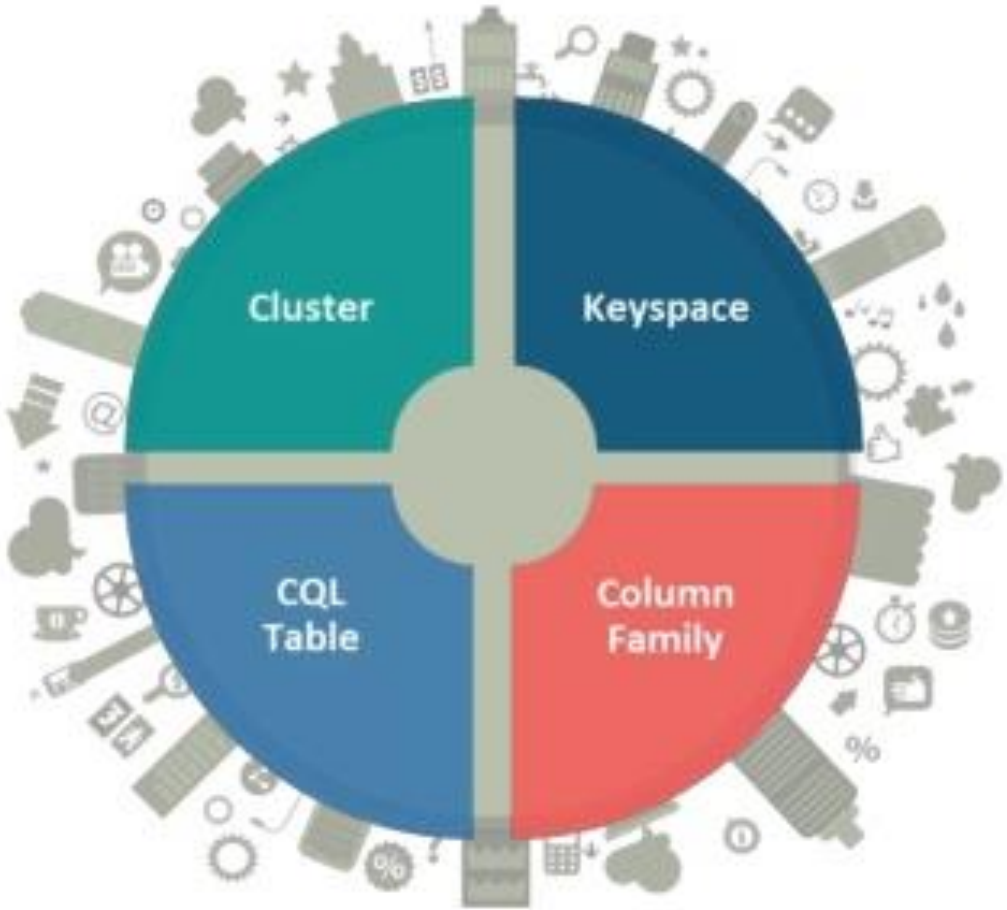
Cassandra in Industry



Cassandra Users in Market


Node
Data Replication
Data Center
1
8
2
SSTables
7
3
Rack
6
4
5
Mem-Table
Cluster
Commit log
Cassandra
ARCHITECTURE

Cassandra Database Elements



Clusters

Cluster is a container for Keyspaces

Keyspace

Keyspace corresponds to database

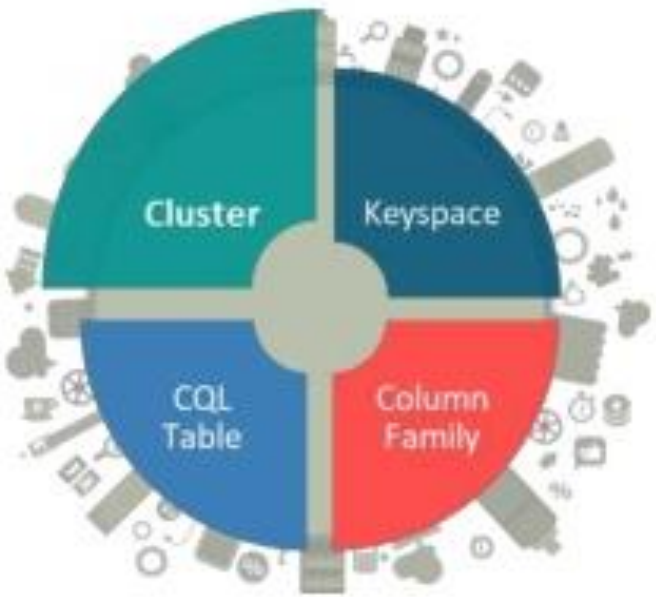
Column Family

Set of rows with a similar structure

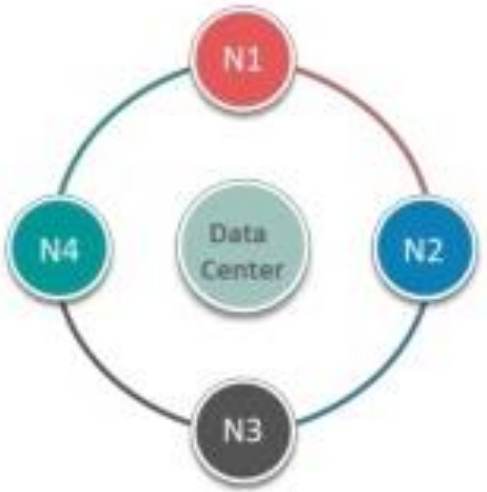
CQL Table

Tables in Cassandra Query Language

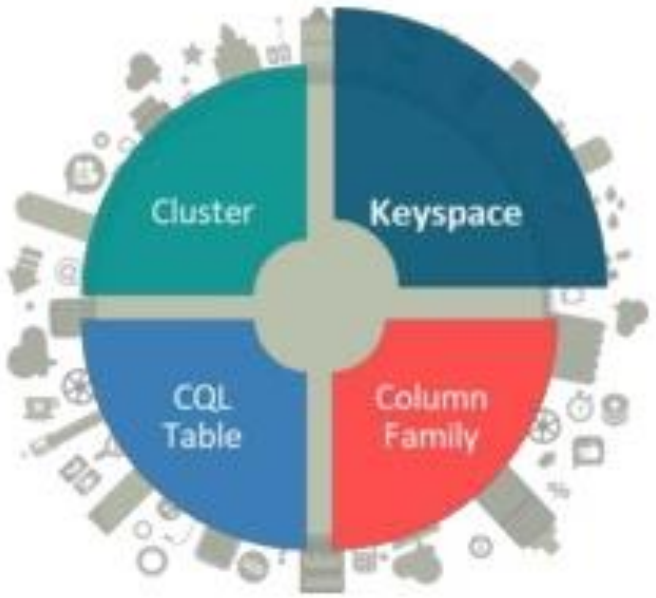
Clusters



- The outermost structure in Cassandra is the *cluster*
- *Cluster is a container for Keyspaces*
- Sometimes called the *ring*, because Cassandra assigns data to nodes in the cluster by arranging them in a ring
- A *node* holds a replica for different range of data

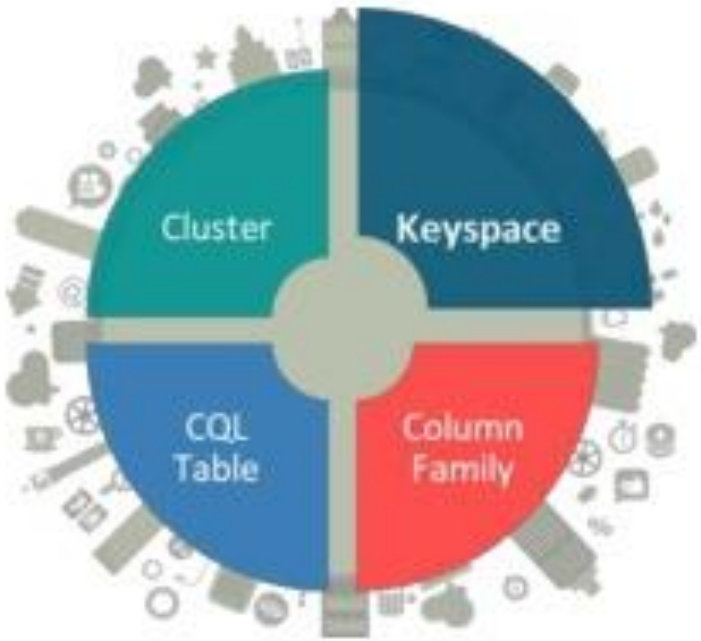


Keyspaces



- A *keyspace* is the outermost container for data in Cassandra
- Like a relational database, *a keyspace has a name and a set of attributes* that define keyspace-wide behaviour
- Keyspace is used to *group Column Families together*

Keyspaces



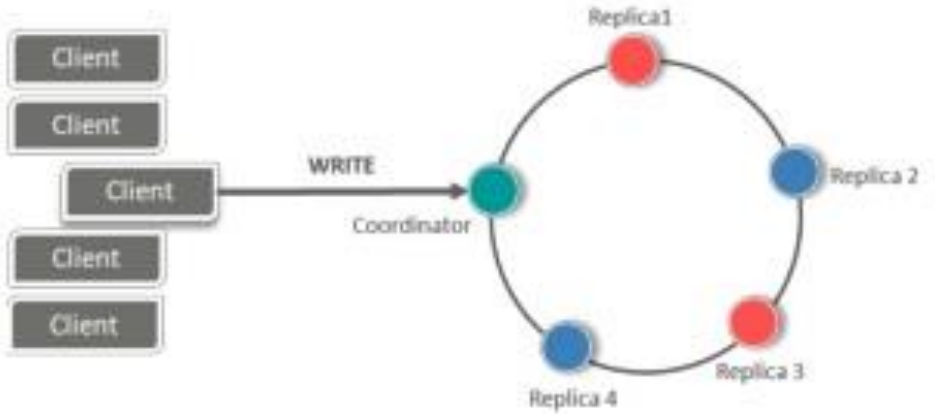

```
File  Edit  View  Search  Terminal  Help
[edureka@localhost ~]$ cqlsh
Connected to Test Cluster at 127.0.0.1:9042.
[cqlsh 5.0.1 | Cassandra 3.11.0 | CQL spec 3.4.4 | Native protocol v4]
Use HELP for help.
cqlsh>
cqlsh>
cqlsh> CREATE KEYSPACE ABC
   ... WITH replication = {'class': 'SimpleStrategy', 'replication_factor': '3'}
   ... AND durable_writes = 'TRUE';
cqlsh>
cqlsh>
```

1. Keyspace Name
2. Replication Strategy
3. Replication Factor
4. Durable Writes

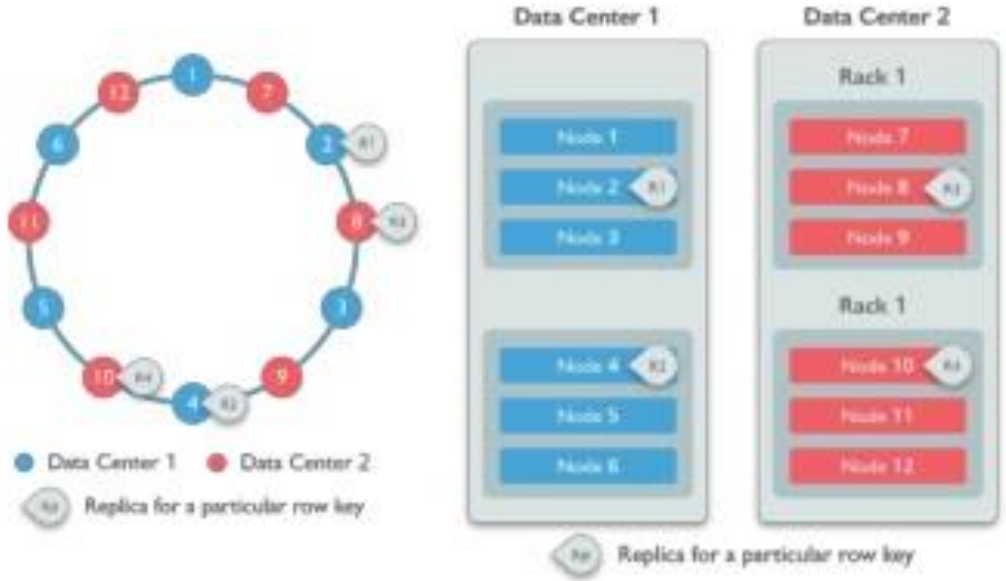
Replica Placement Strategy

- The *replica placement strategy* refers to how the replicas will be placed in the ring
- There are different strategies that ship with Cassandra for determining *which nodes will get copies of which keys*
- There are mainly two types of Strategies:
  - *Simple Strategy*
  - *Network Topology Strategy*

Replica Placement Strategy

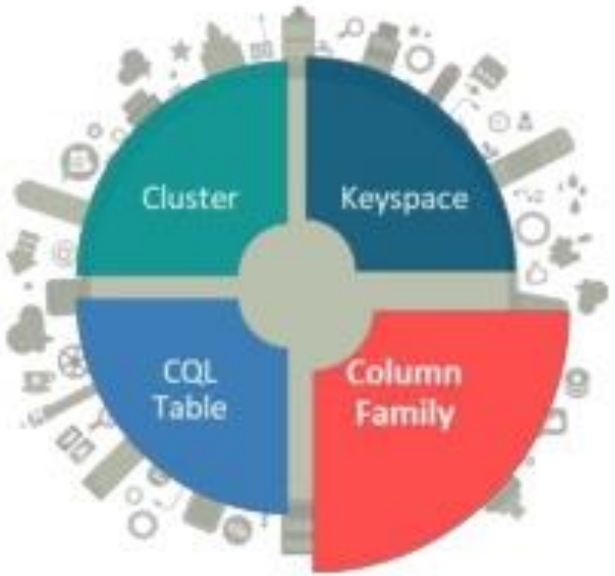


SIMPLE STRATEGY

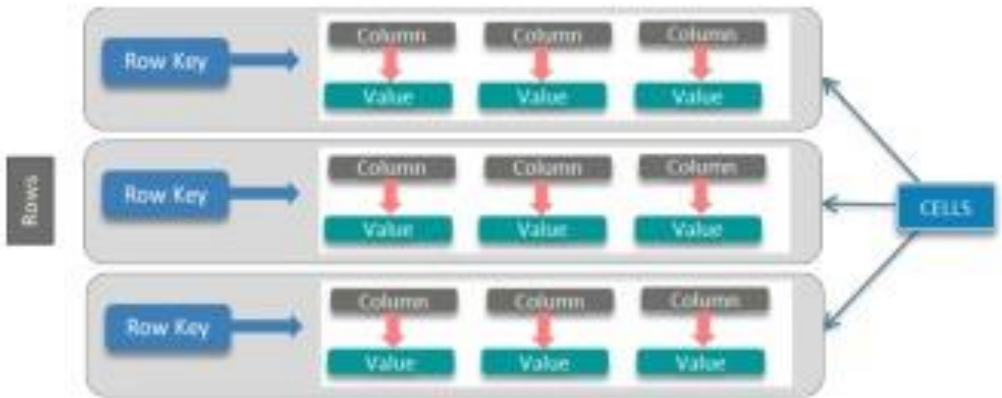


NETWORK TOPOLOGY STRATEGY

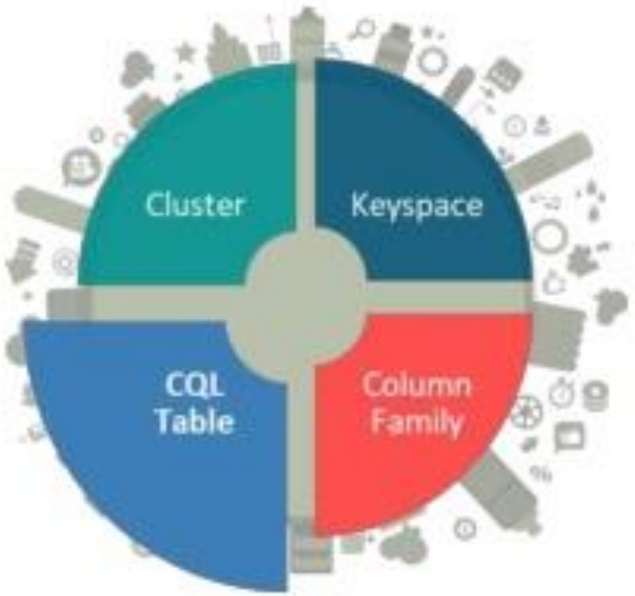
Column Family



- A column family is a container *for an ordered collection of rows, each of which is itself an ordered collection of columns*
- We can freely *add any column to any column family at any time,* depending on your needs



CQL Table



- A CQL Table is a *column family*
- CQL tables provide *two-dimensional view of a column family*, which contains potentially multi-dimensional data
- CQL table and column family are largely interchangeable terms

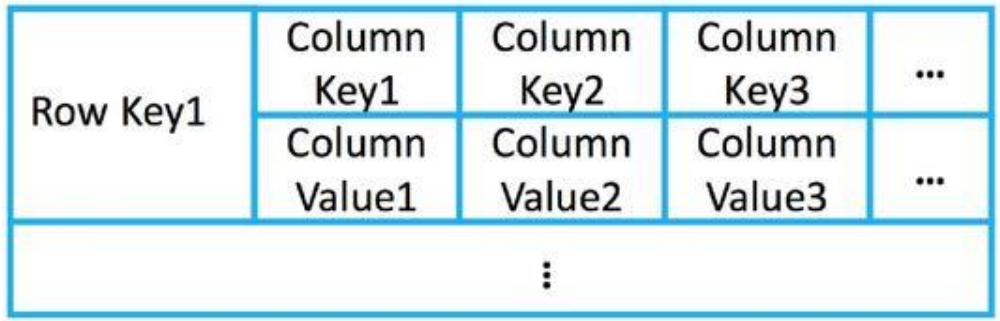
```
File  Edit  View  Search  Terminal  Help
cqlsh:abc> CREATE TABLE race_winners (
       ...     race_name text,
       ...     race_position int,
       ...     cyclist_name text,
       ...     PRIMARY KEY (race_name, race_position));
cqlsh:abc>
cqlsh:abc>
```

Why Cassandra?

It has a flexible data model

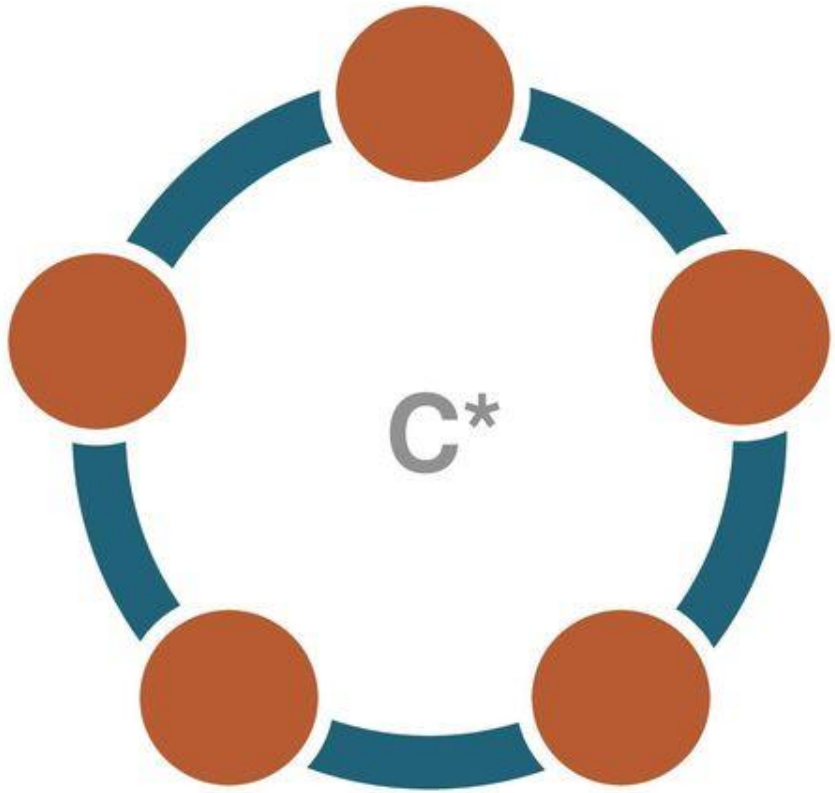
Tables, wide rows, partitioned and distributed

- ✓ Data
- ✓ Blobs (documents, files, images)
- ✓ Collections (Sets, Lists, Maps)
- ✓ UDTs



Access it with CQL ← familiar syntax to SQL

It’s a Cluster of Nodes



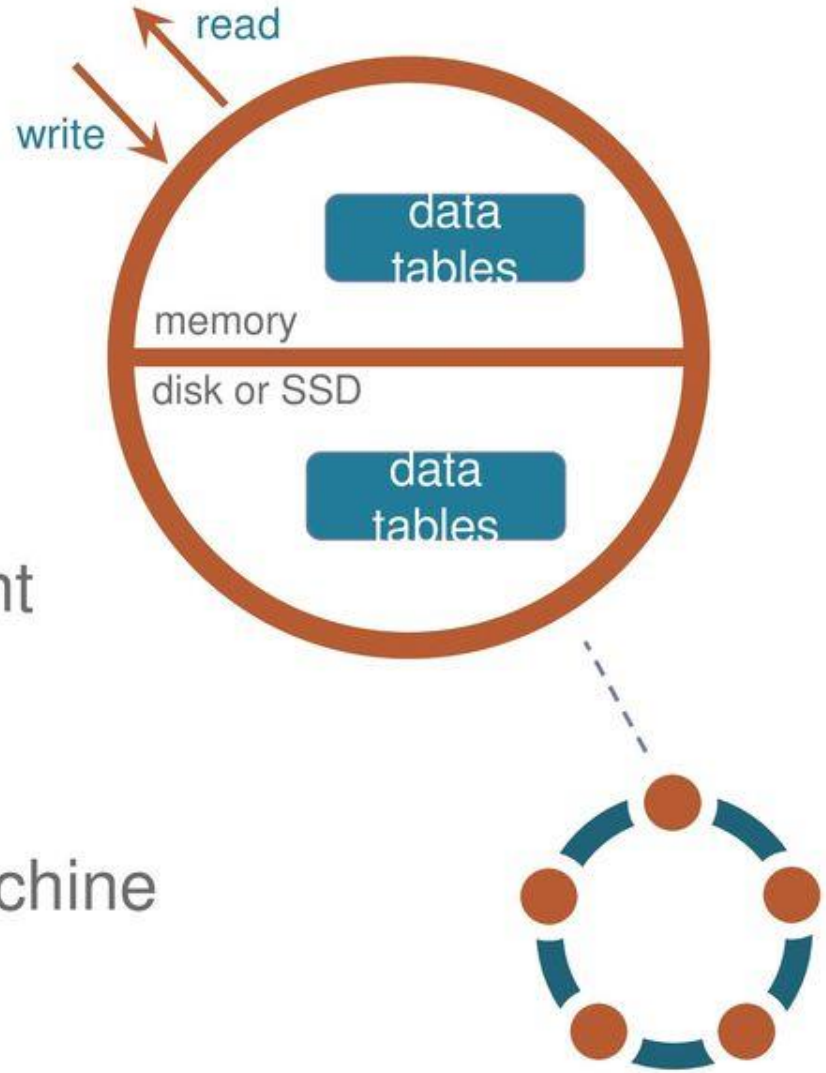
Each Cassandra node is…

Fully functional database

Very fast (low latency)

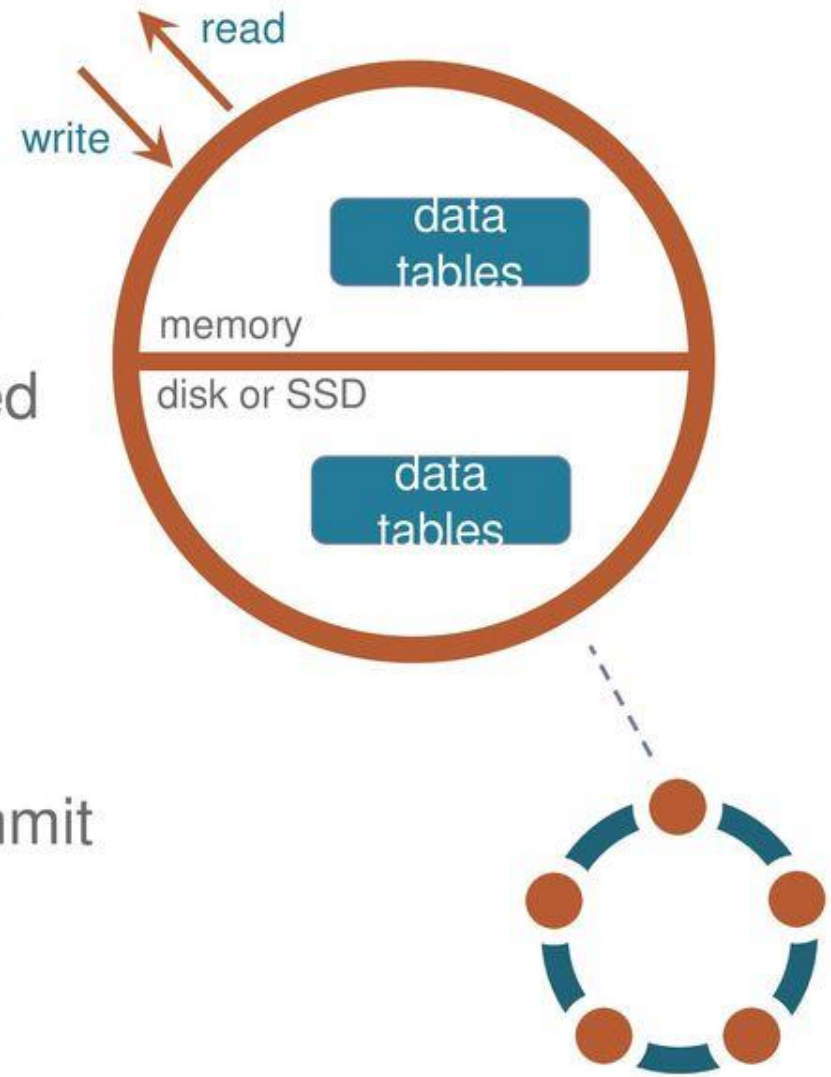
In-memory and/or persistent storage

One machine or Virtual Machine



Why is Cassandra so fast?

Low Latency Nodes
Distributed Workload



Writes

- Durable write to log file (fast)
- No db file read or lock needed

Reads

- Get data from memory first
- Optimize storage layer IO
- No locking or two phase commit

A Cassandra Cluster

Nodes in a peer-to-peer cluster

No single point of failure

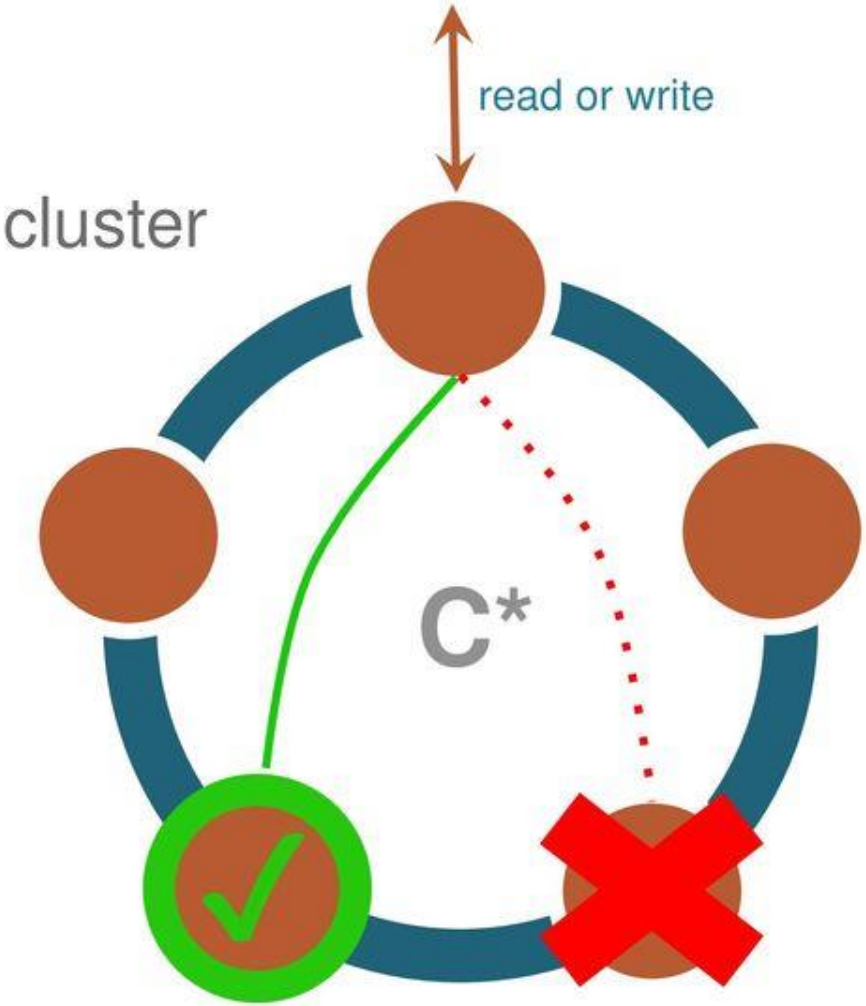
Built in data replication

Data is always available

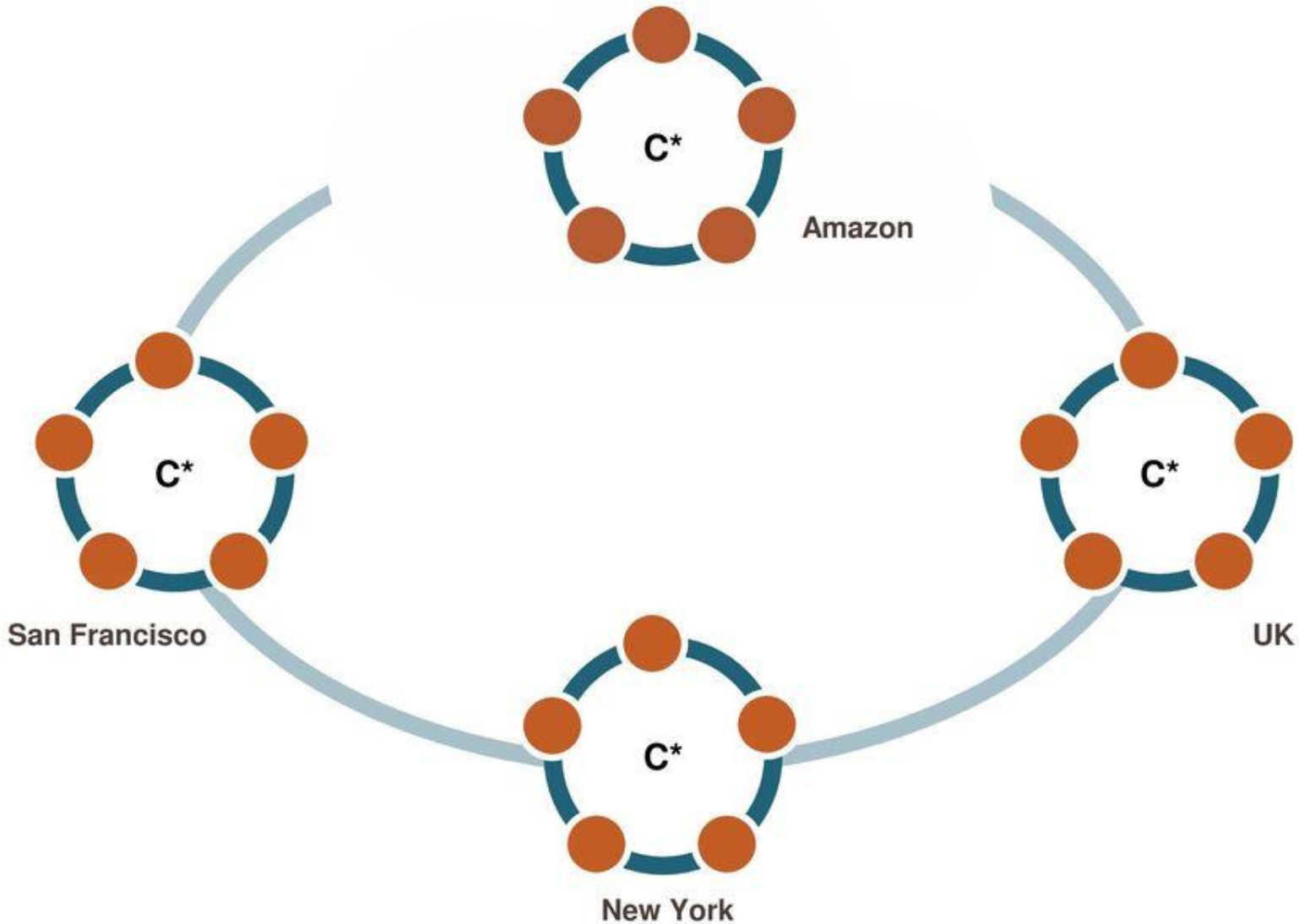
100% Uptime

Across data centers

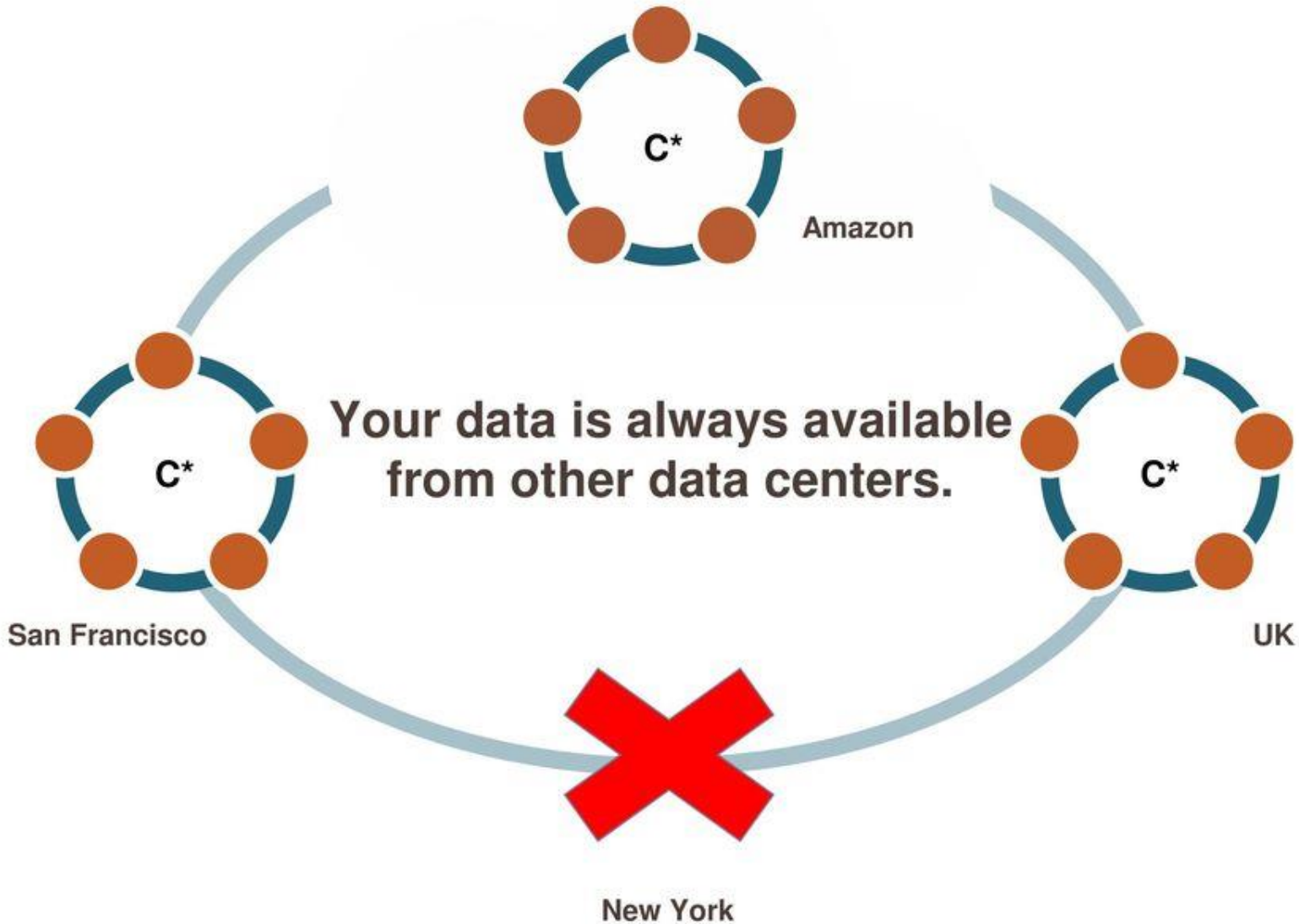
Failure avoidance



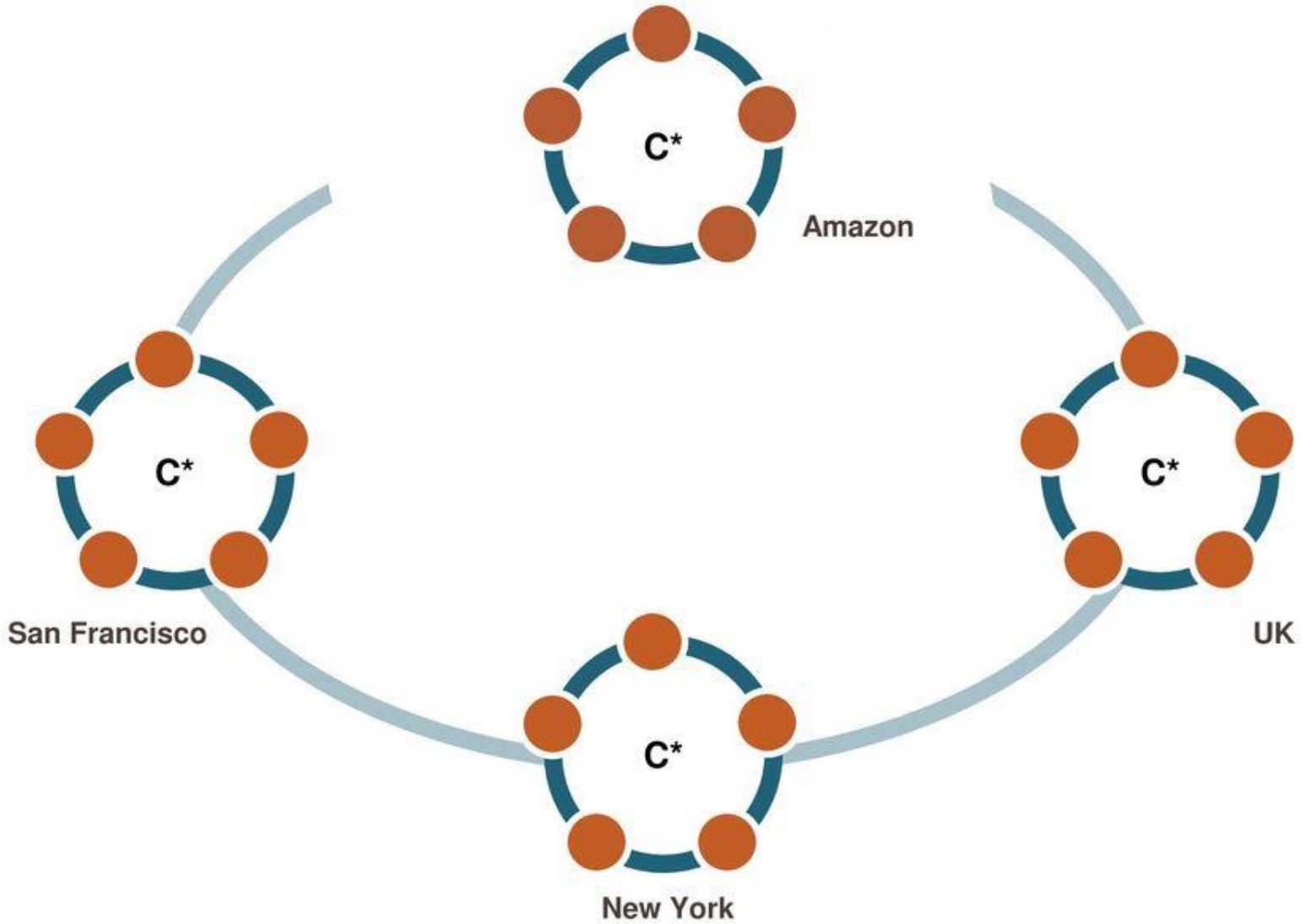
Multi-data center deployment



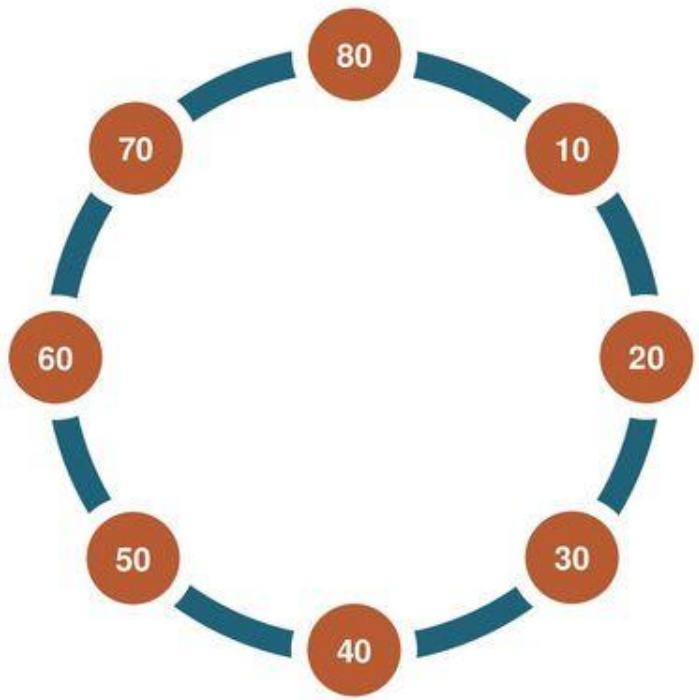
If a data center goes offline…



…and recovers automatically



Cassandra Cluster Architecture



Each Node ~ box or VM

(technically it's a JVM)

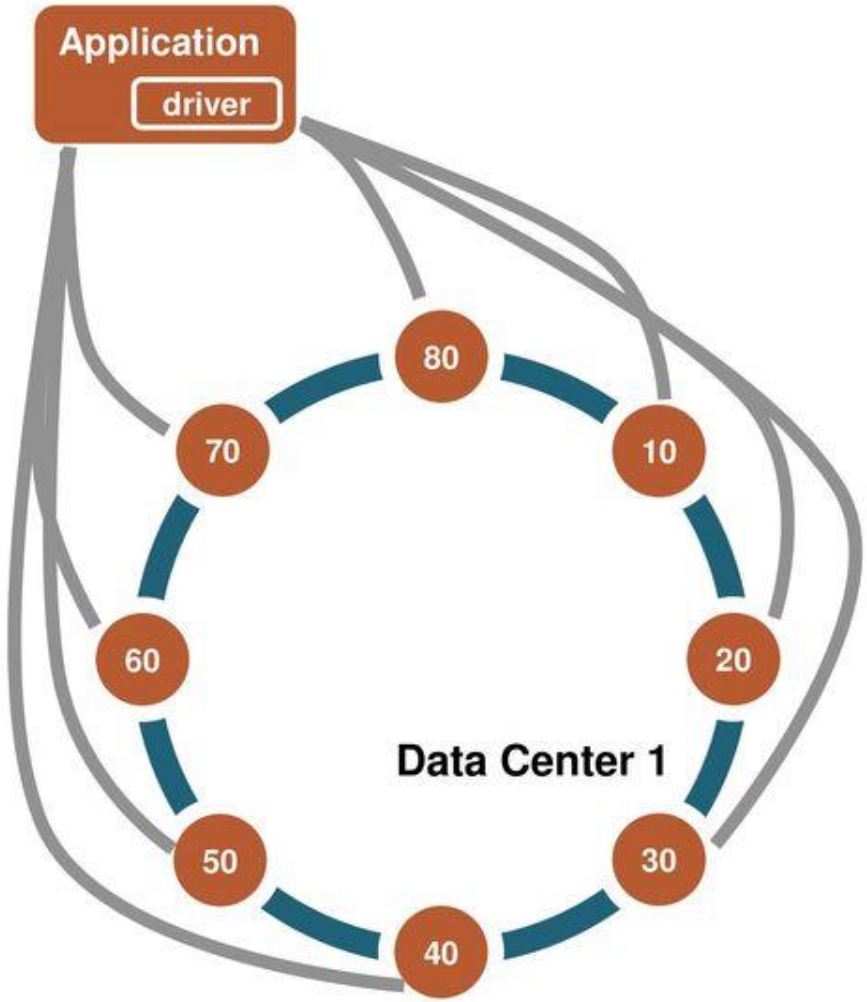
Each node has same Cassandra database functionality

System/hardware failures happen

Snitch – topology (DC and rack)

Gossip – state of each node

Transaction Load Balancing



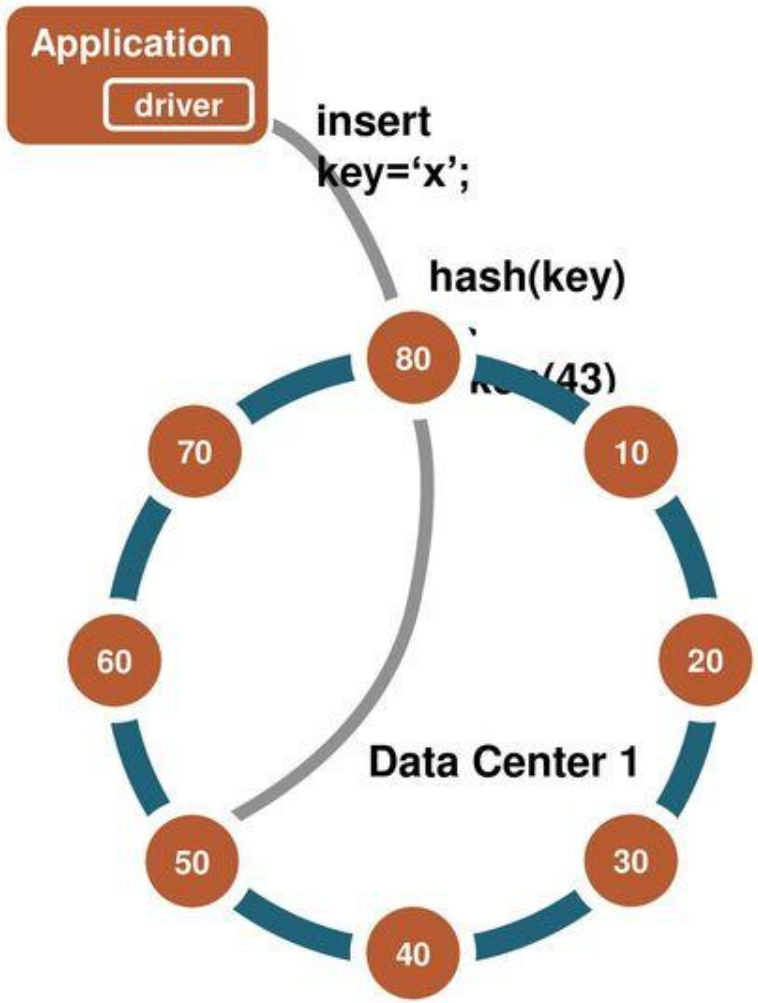
Driver manages connection pool

Driver has load balancing policies that can be applied

Each transaction has a different connection point in the cluster

Asynch operations are faster

Data Partitioning



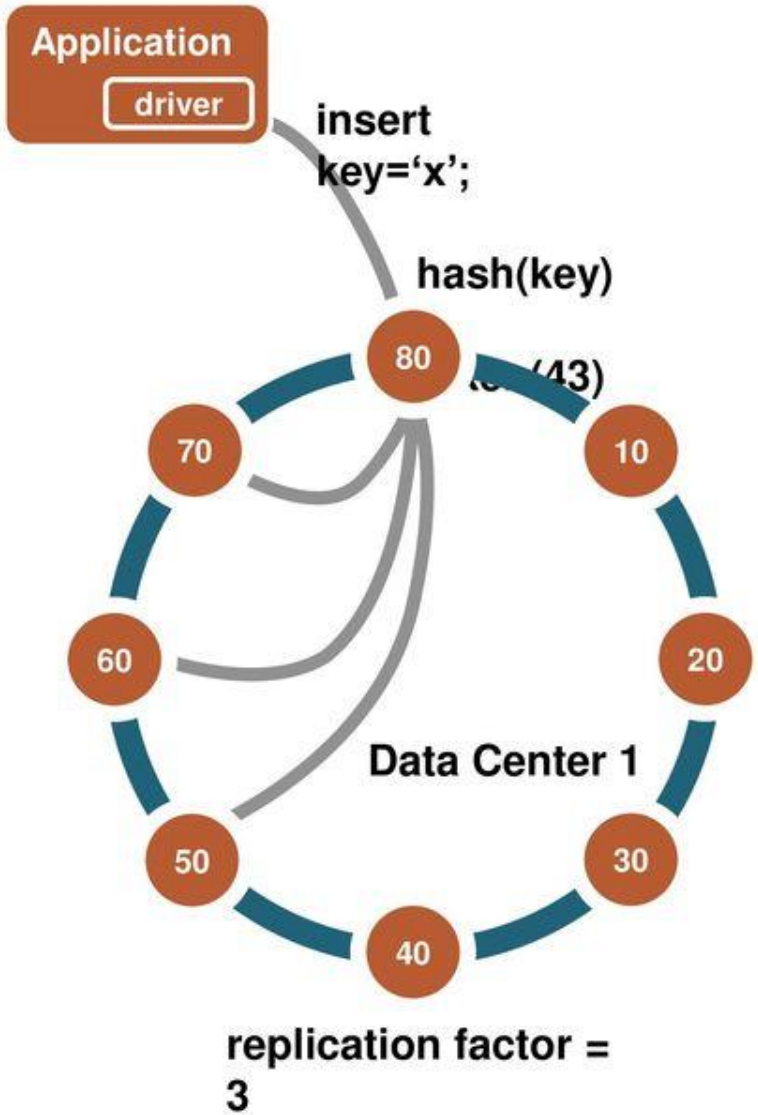
Driver selects the Coordinator node per operation via load balancing

Partitioned token ranges

- Random distribution
- Murmur3
- No hot spots

Each entire row lives on a node

Replication



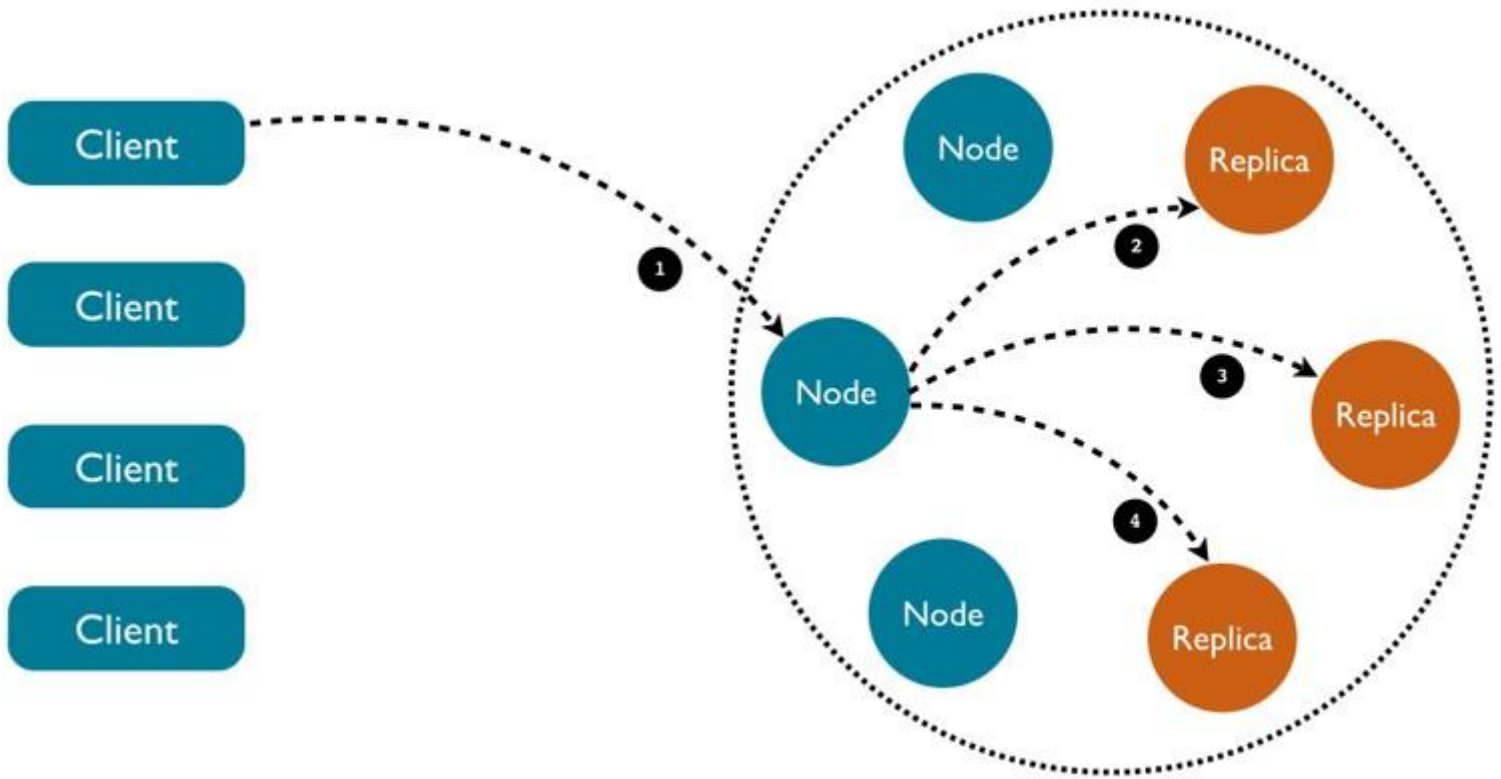
Replication Factor = # of copies

All replication operations in parallel

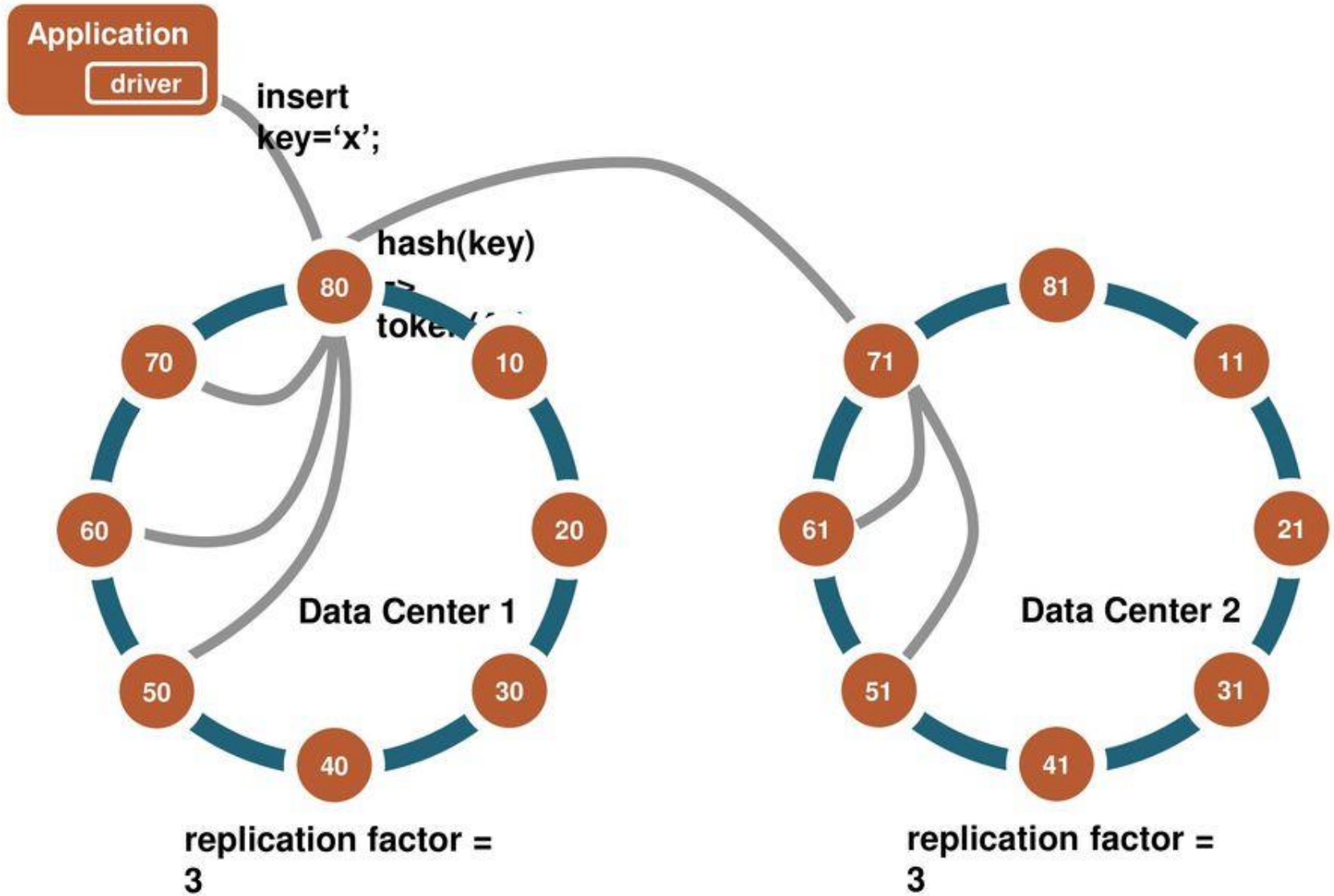
"Eventual" = micro- or milliseconds

No master or primary node

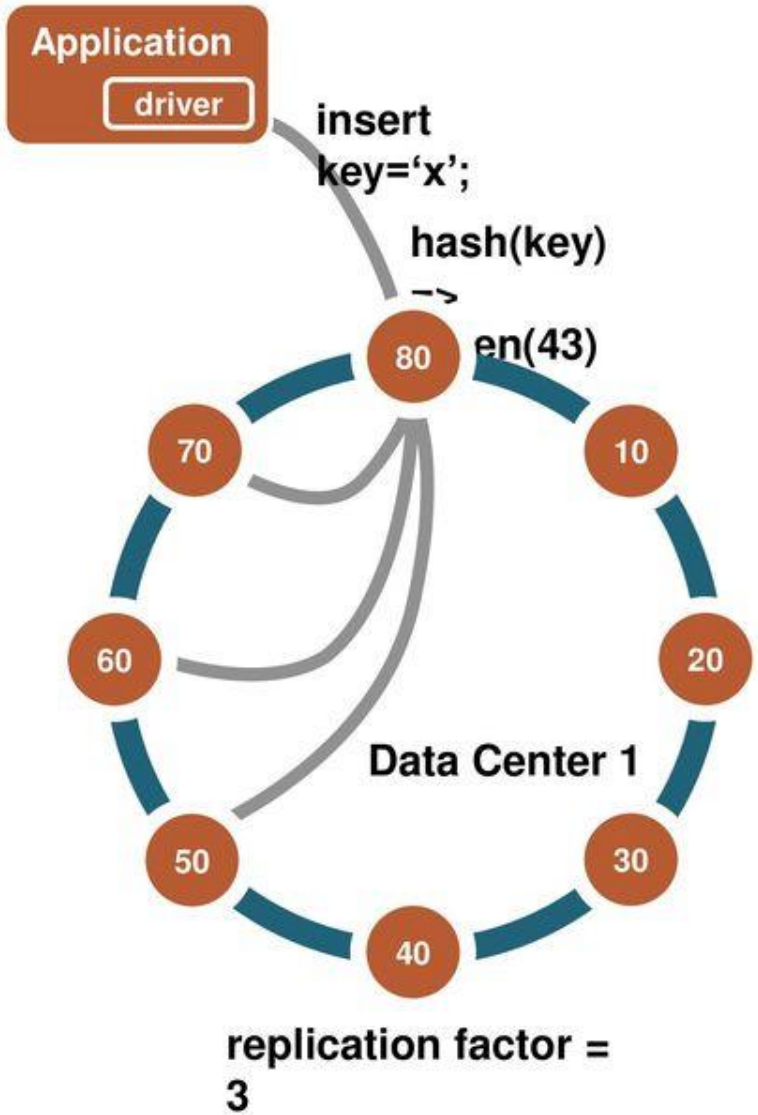
Each node acknowledges the op


Client
Client
Client
Client
1
Node
Replica
2
Node
3
Replica
4
Node
Replica

Multi-Data Center Replication



Consistency



Replication Factor = ?

Consistency Level = # of nodes to acknowledge an operation

read and write "consistency" per operation

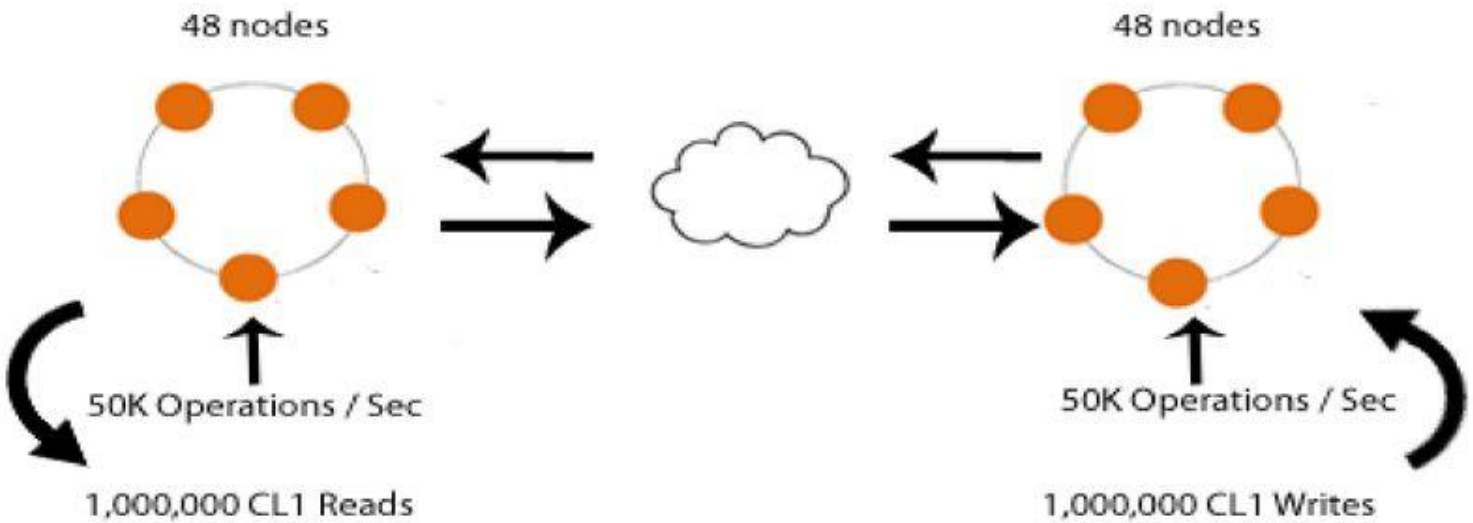
CL(write) + CL(read) > RF

→

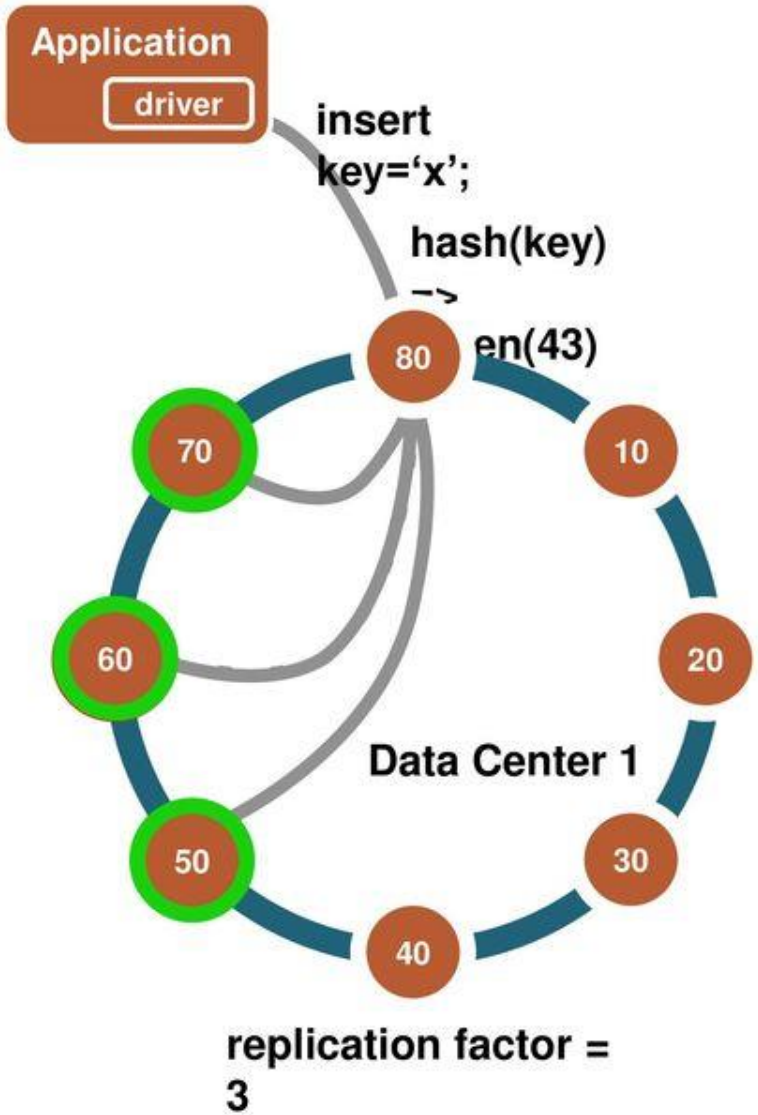
consistency

If needed, tune consistency for performance

Netflix Replication Experiment



Slow Node Anti-Entropy: Hints



IFF Node 60 is:

Slow enough to timeout ops

Down for a short time

Then…

Node 80 holds the op missed by 60

held ops are called Hints

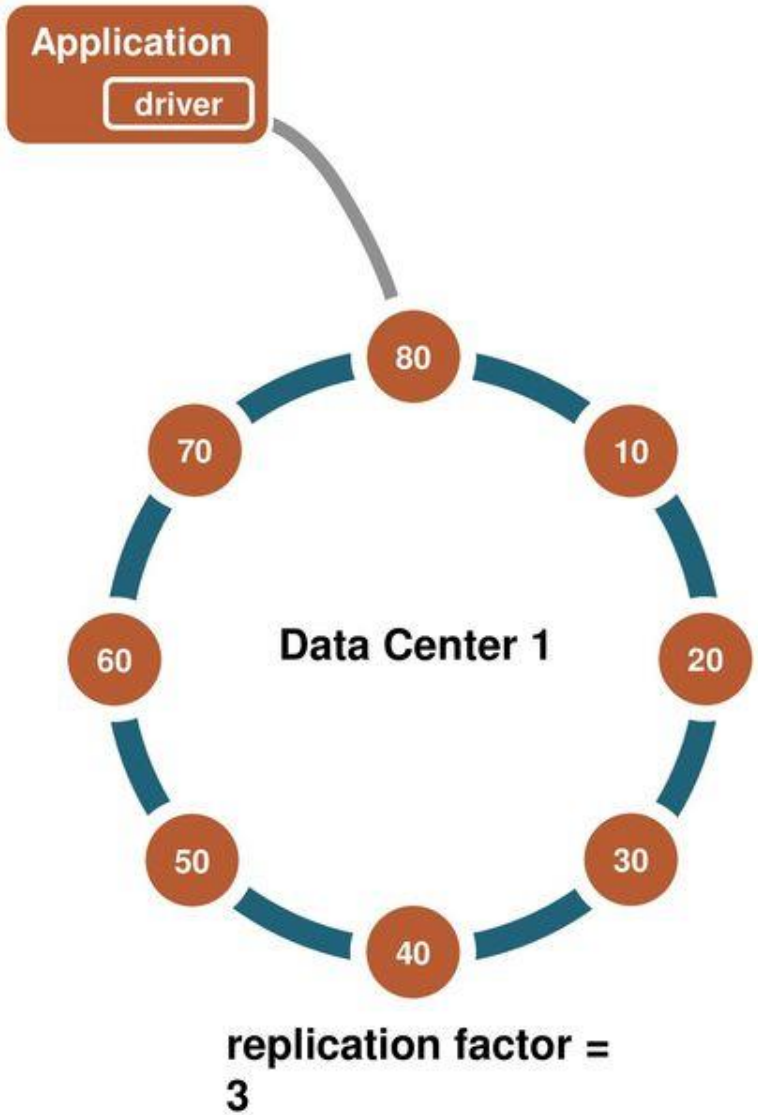
When node 60 becomes responsive

Node 80 sends the hints as ops

…until node 60 is synchronized

Read Repair

What if I need more nodes?



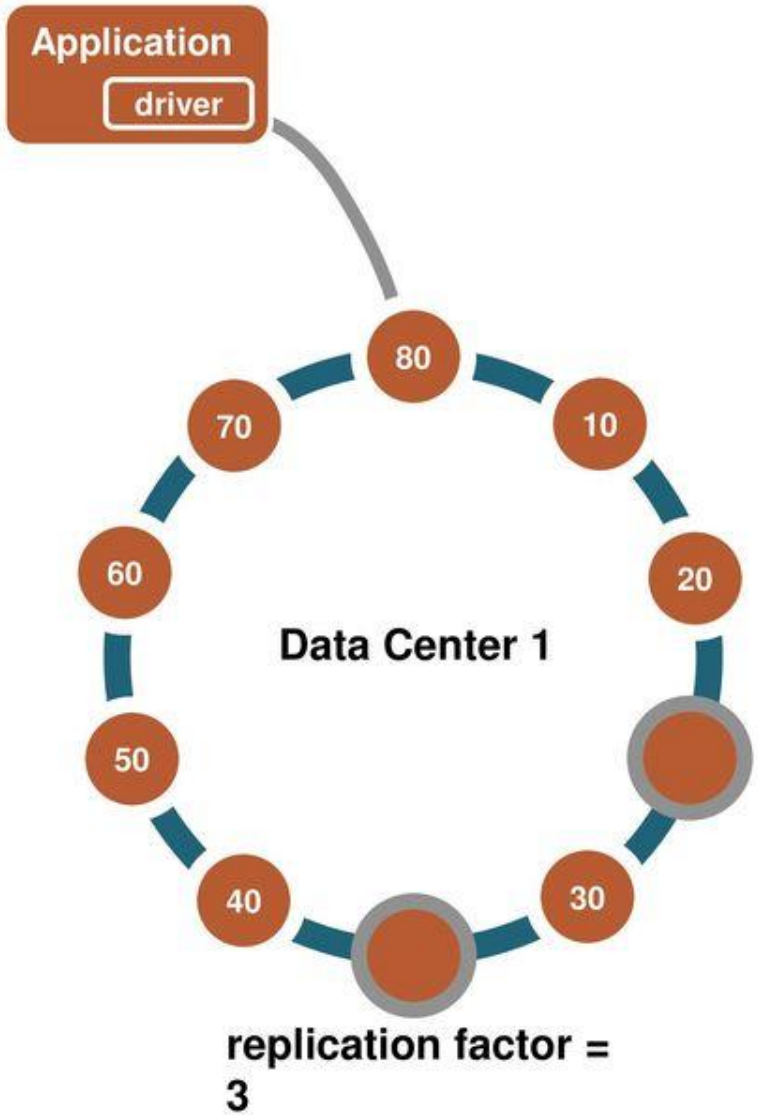
Data set size is growing

Need more TPS

- Client application demand
- Hardware limits being reached
- Looking for lower latency

Moving some tables to in-memory

Cluster Expansion: Add Nodes

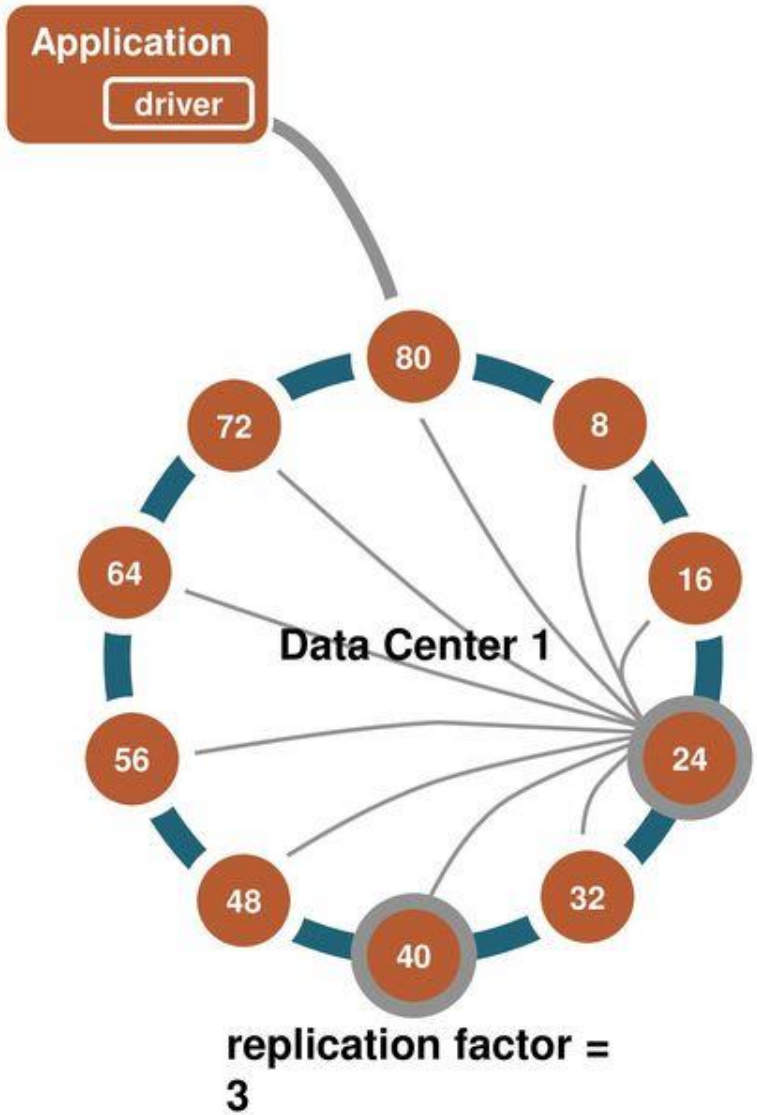


Introduce new nodes to cluster

Nodes do not yet own any token ranges (data)

Cluster operates as normal

Cluster Expansion: Rebalance



Rebalance = redistribution of token ranges around the cluster

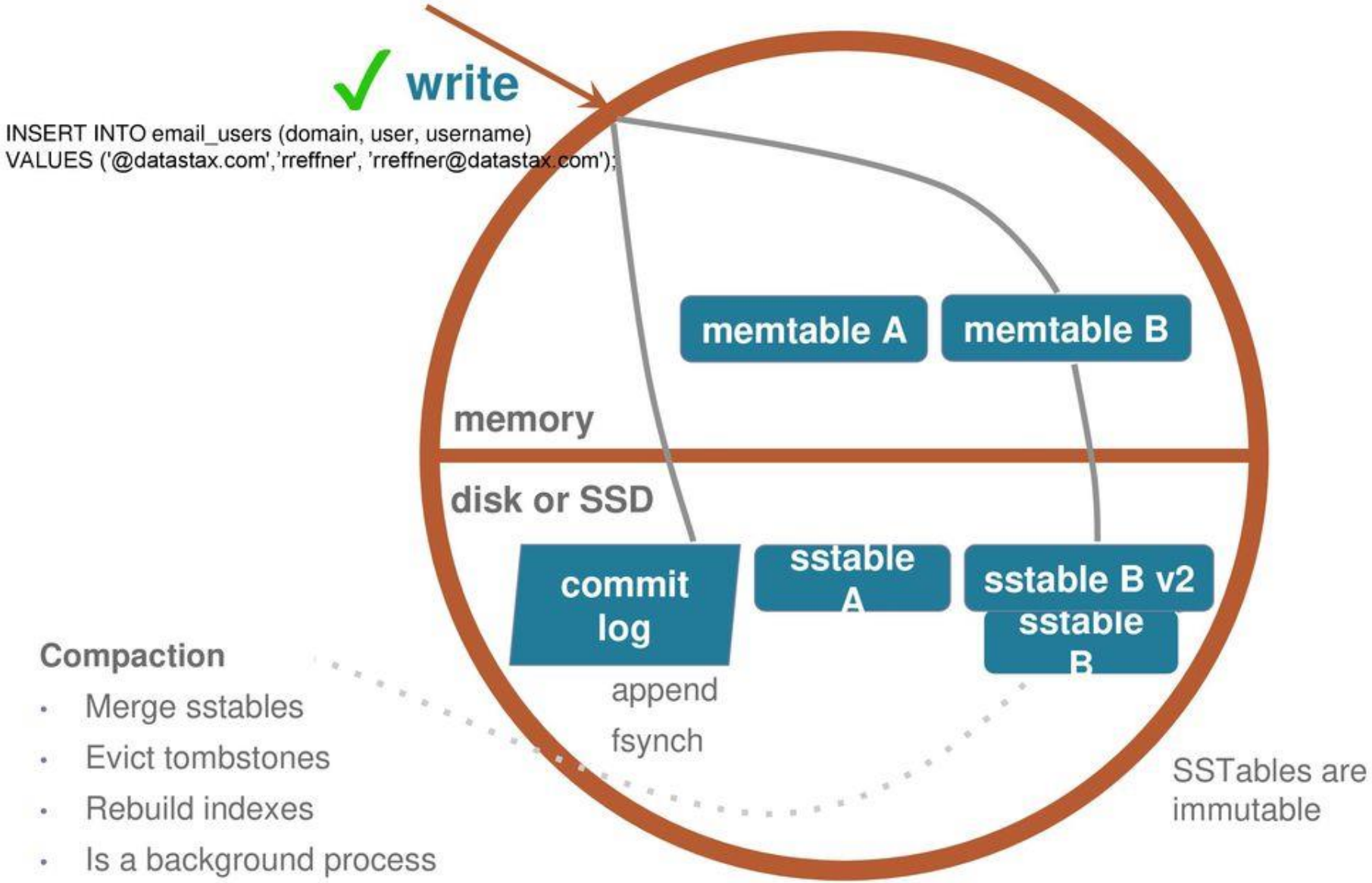
Data is streamed in small chunks to synchronize the cluster

- see “Vnodes”
- minimizes streaming
- distributes rebalance workload

New nodes begin taking ops

Cleanup = reclaims space on nodes dedicated to tokens no longer owned

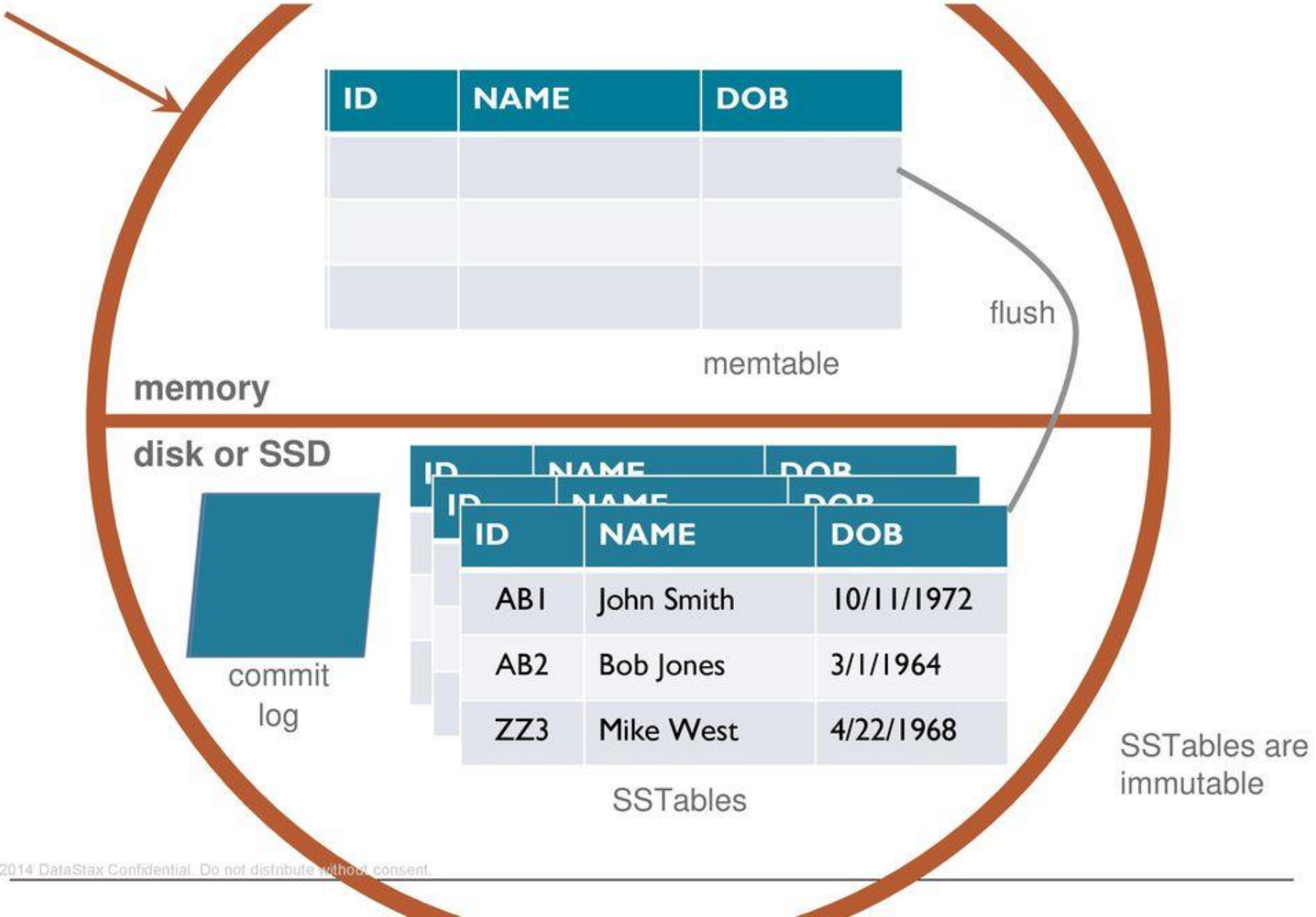
Write Path



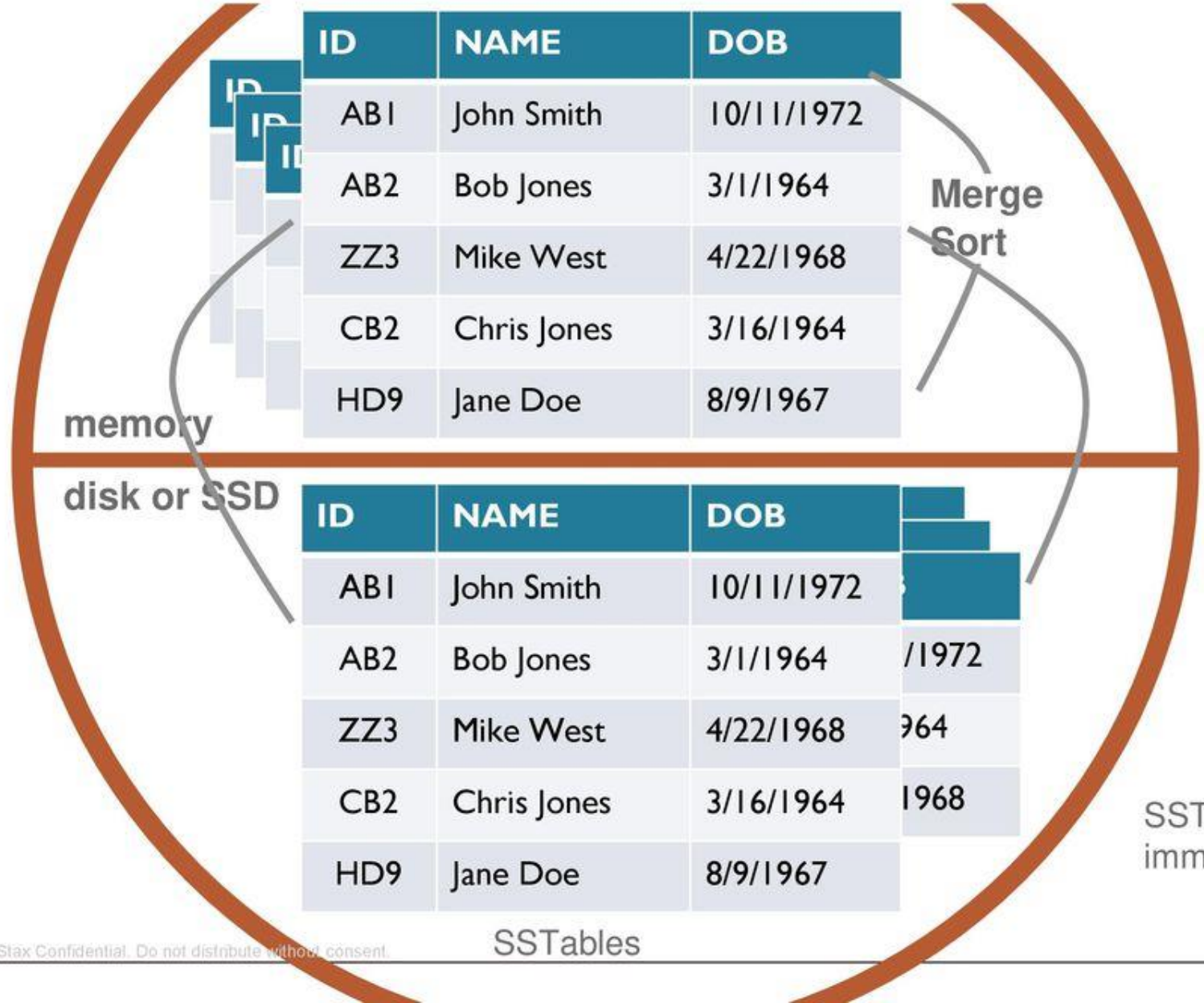
Compaction

- Merge sstables
- Evict tombstones
- Rebuild indexes
- Is a background process

Memtable Flush During Write Load



Compaction During Write Load



Compaction And Tombstones

ID	NAME	DOB
BB1	John Waters	
CB2		4/20/1964
NN3	Jim West	4/22/1958

ID	NAME	DOB
BB1		11/11/1974
CB2	Chris Jones	~~3/16/1964~~
~~NN3~~	~~Jim West~~	~~4/22/1958~~

ID	NAME	DOB
AB1	John Smith	10/11/1972
AB2	Bob Jones	
ZZ3	Mike West	4/22/1968

Note: This is a logical representation, not the SSTable physical file format.

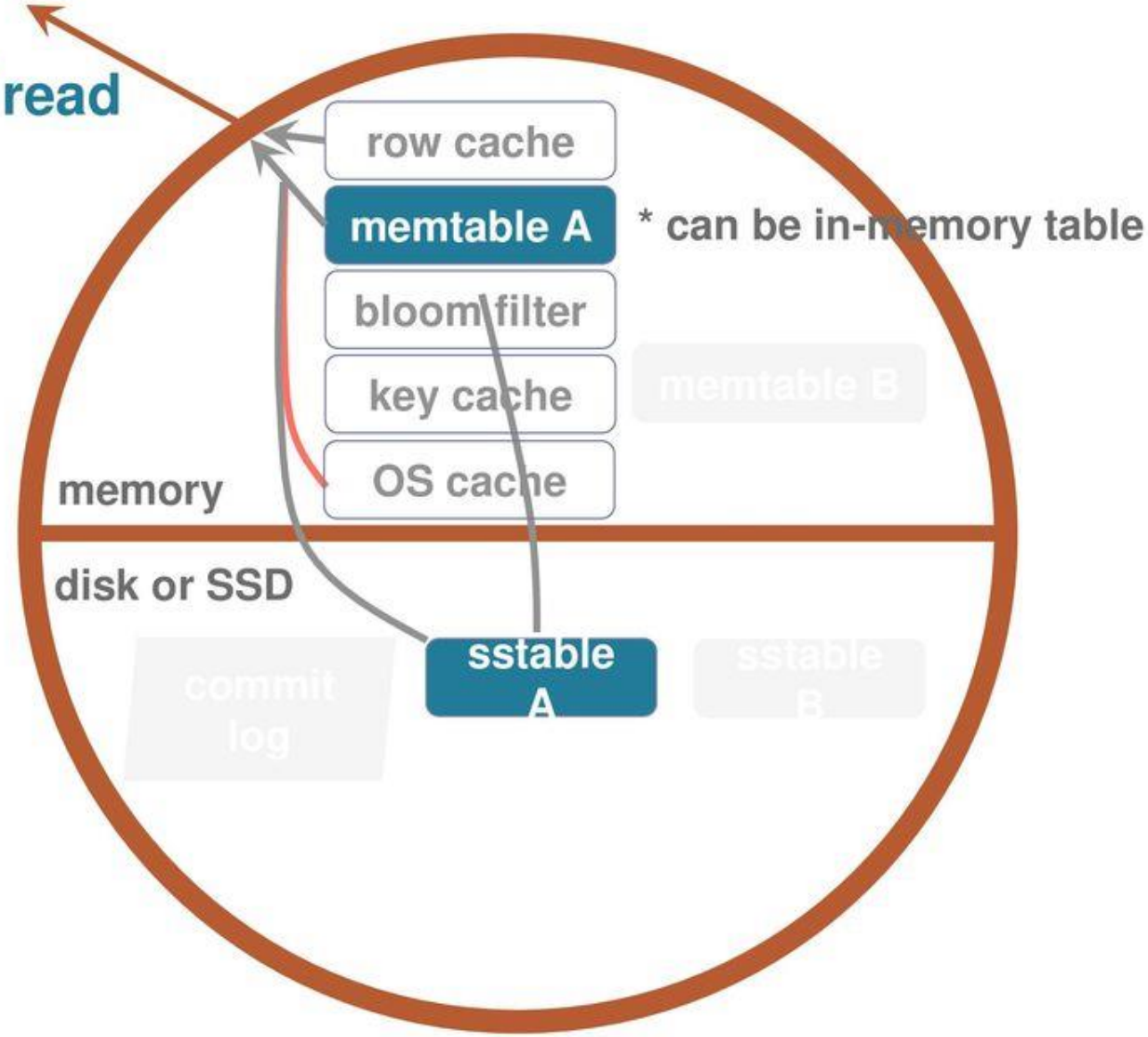
ID	NAME	DOB
AB1	John Smith	10/11/1972
AB2	Bob Jones	
ZZ3	Mike West	4/22/1968
CB2	Chris Jones	
BB1	John Waters	11/11/1974

SSTables are immutable

Read Path

✓✓✓ read

```
SELECT * FROM email_users
WHERE domain= '@datastax.com'
AND user = 'rreffner@datastax.com';
```



Data Modeling Best Practices

Start with your data access patterns

- What does your app do?
- What are your query patterns?

Optimize for

- Fast writes and reads at the correct consistency
- Consider the results set, order, group, filtering
- Use TTL to manage data aging
- 1 Query = 1 Table
  - Each SELECT should target a single row (1 seek)
  - Range queries should traverse a single row (efficient)

Denormalization Is Expected

Remember: You're *not* optimizing for storage efficiency

You are optimizing for *performance*

Do this:

- Forget what you've learned about 3rd normal form
- Repeat after me… "slow is down", "storage is cheap"
- Denormalize and do parallel writes!

Don't to this:

- Client side joins
- Reads before writes

Denormalization Is Expected

Remember: You're *not* optimizing for storage efficiency

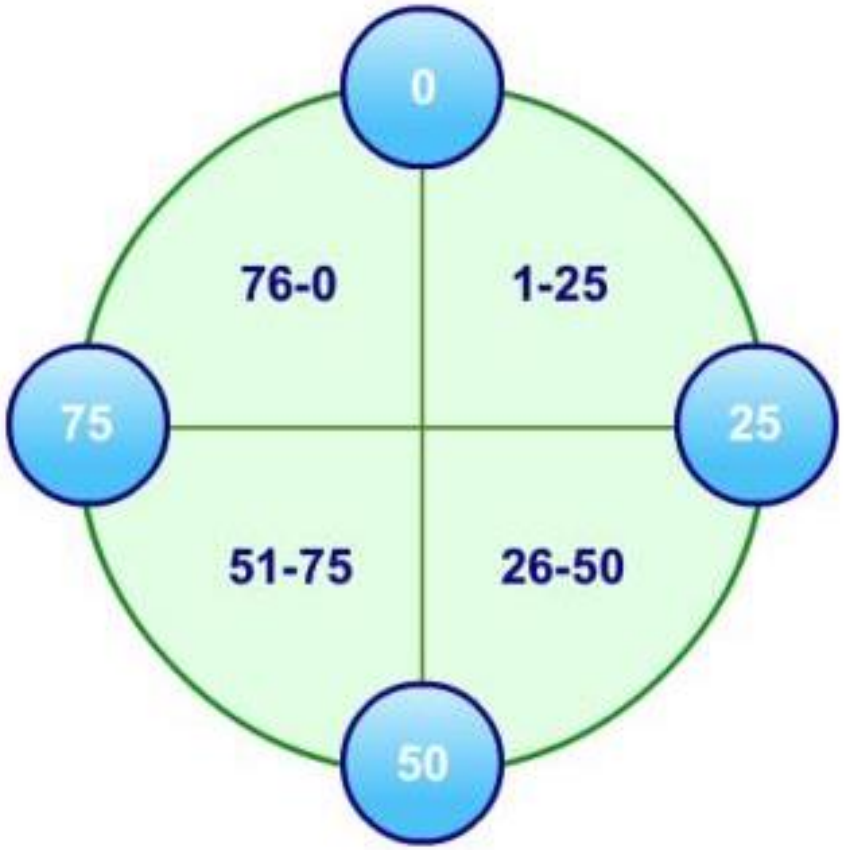
You are optimizing for *performance*

Do this:

- Forget what you've learned about 3rd normal form
- Repeat after me… "slow is down", "storage is cheap"
- Denormalize and do parallel writes!

Don't to this:

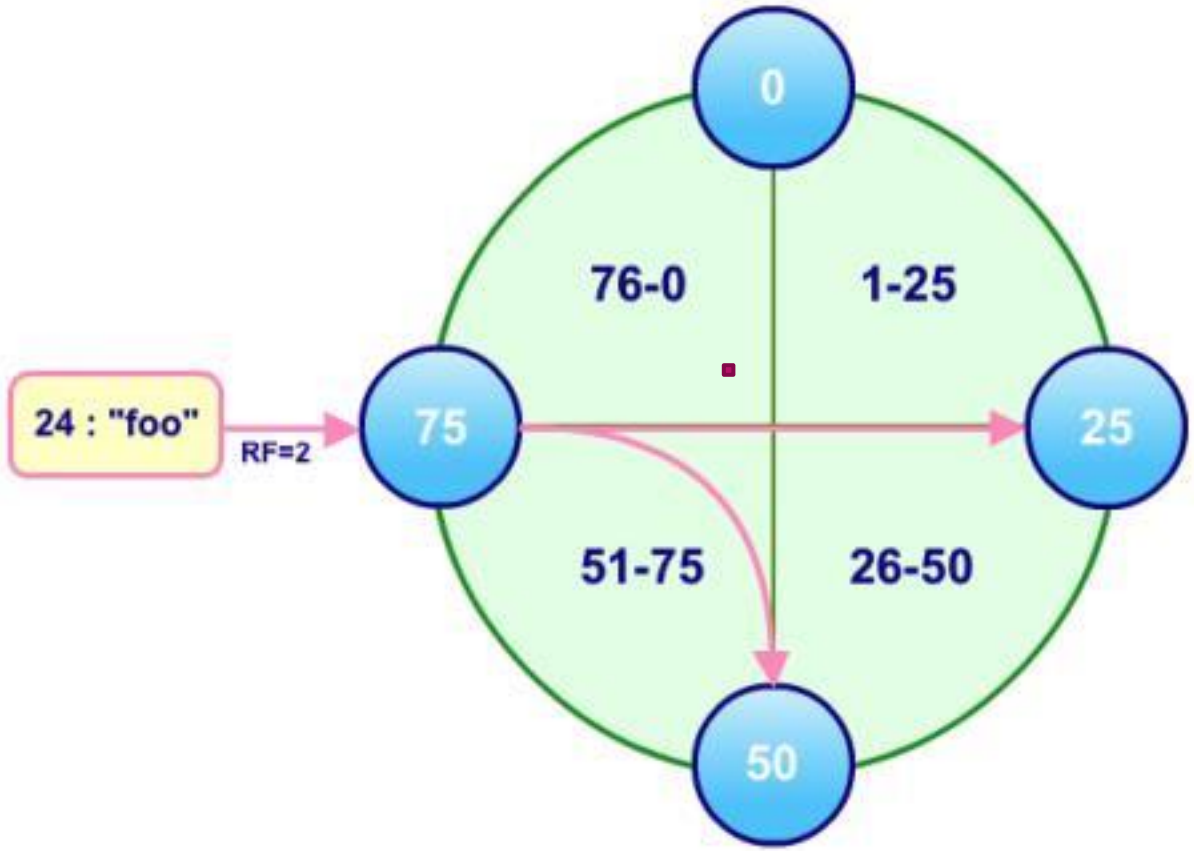
- Client side joins
- Reads before writes



Each physical node is assigned a token

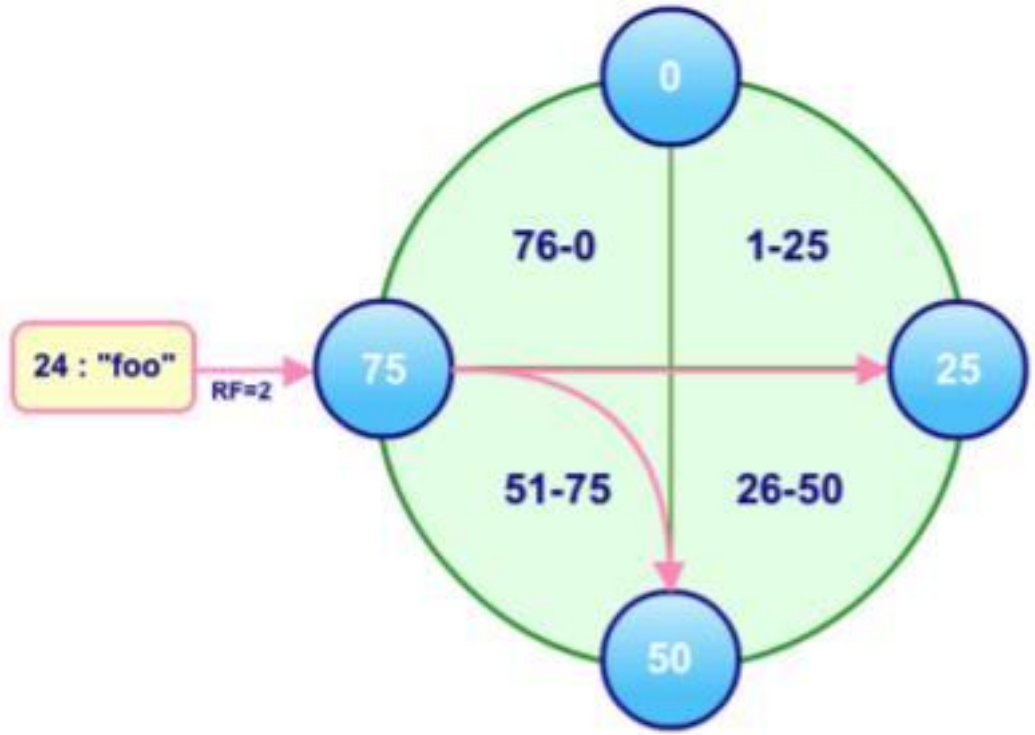
Nodes own the range from the previous token

Cassandra Write Path



The coordinator will send the update to two nodes, starting at the owning node and working clockwise

Cassandra Write Path

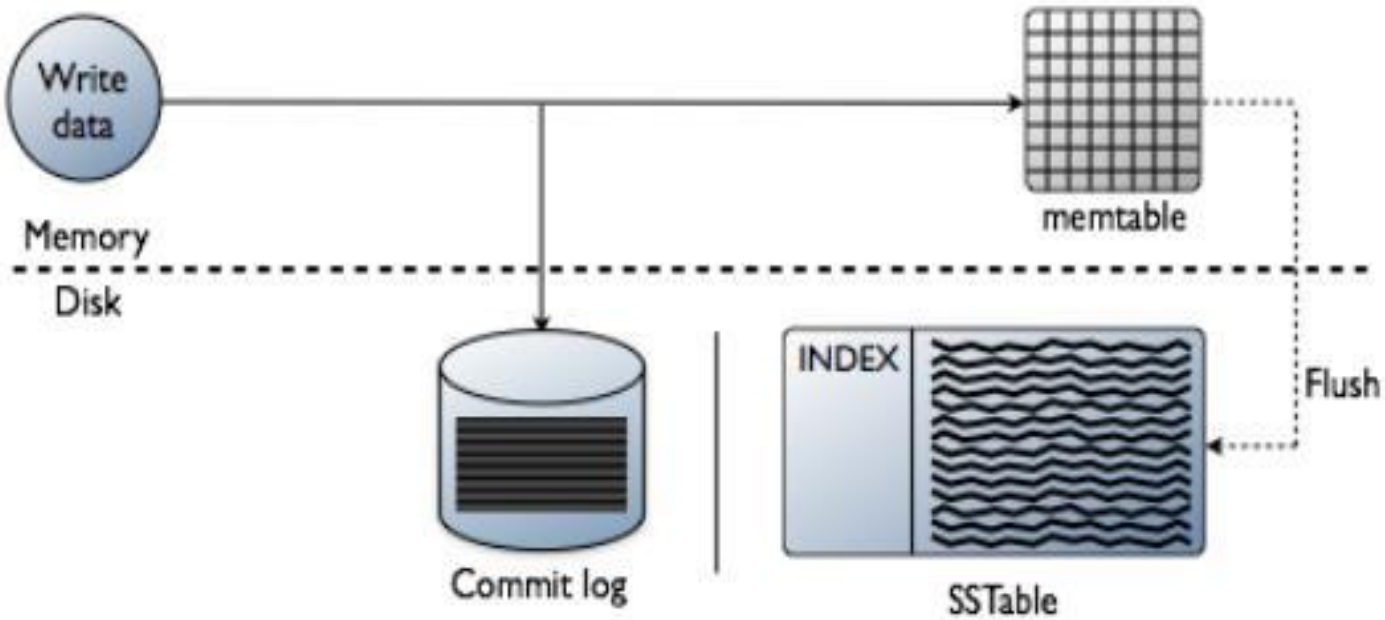


128-bit hash used to compute partition key

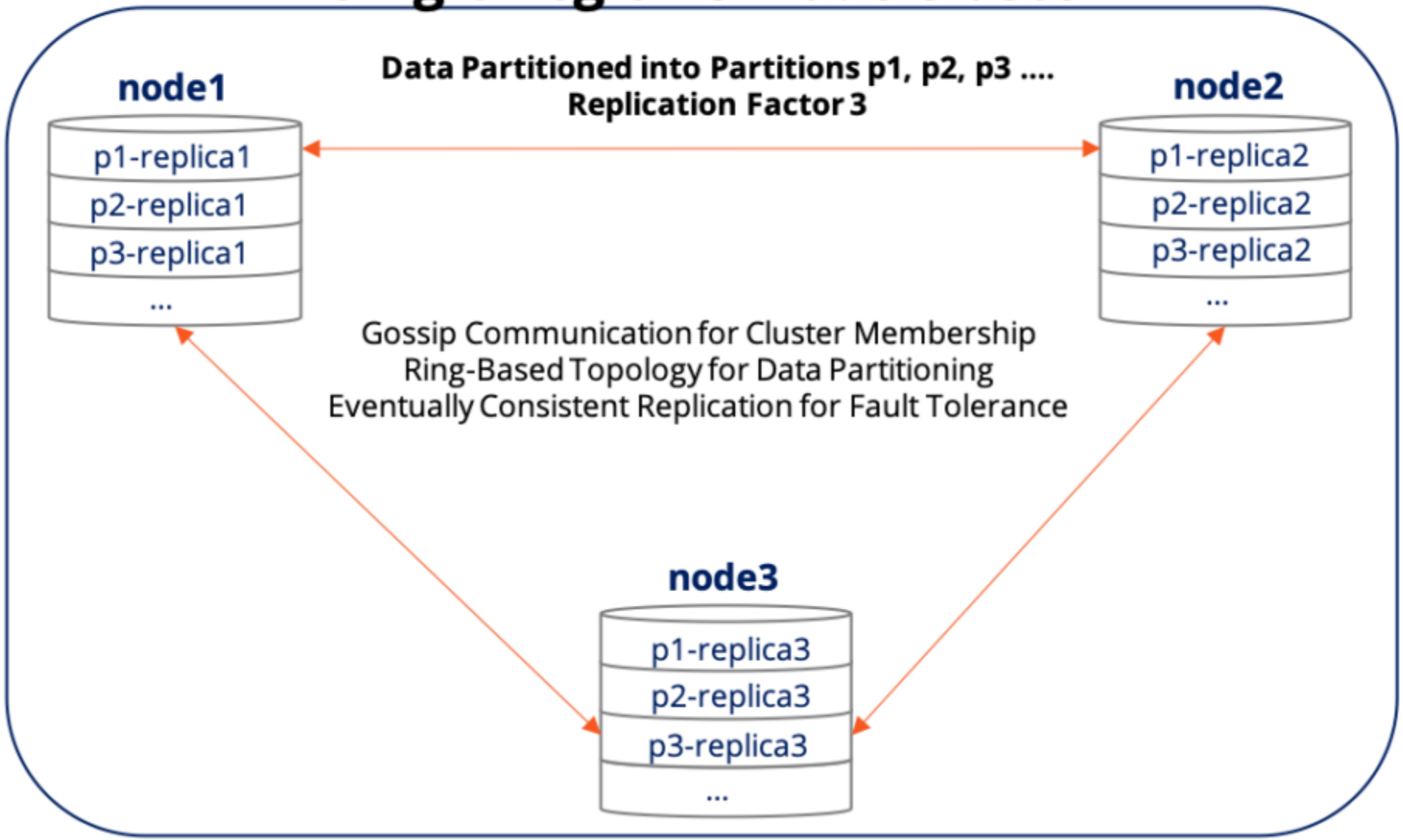
Keys are therefore distributed randomly around the ring

If Unavailable - Hinted Handoff

Cassandra Write Path



- SSTables are sequential and immutable
- Data may reside across SSTables
- SSTables are periodically compacted together


Apache cassandra™
Single Region 3 Node Cluster
Data Partitioned into Partitions p1, p2, p3 ....
Replication Factor 3
node1
p1-replica1
p2-replica1
p3-replica1
...
node2
p1-replica2
p2-replica2
p3-replica2
...
Gossip Communication for Cluster Membership
Ring-Based Topology for Data Partitioning
Eventually Consistent Replication for Fault Tolerance
node3
p1-replica3
p2-replica3
p3-replica3
...

Cassandra Write Data Flows

Single Region, Multiple Availability Zone

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

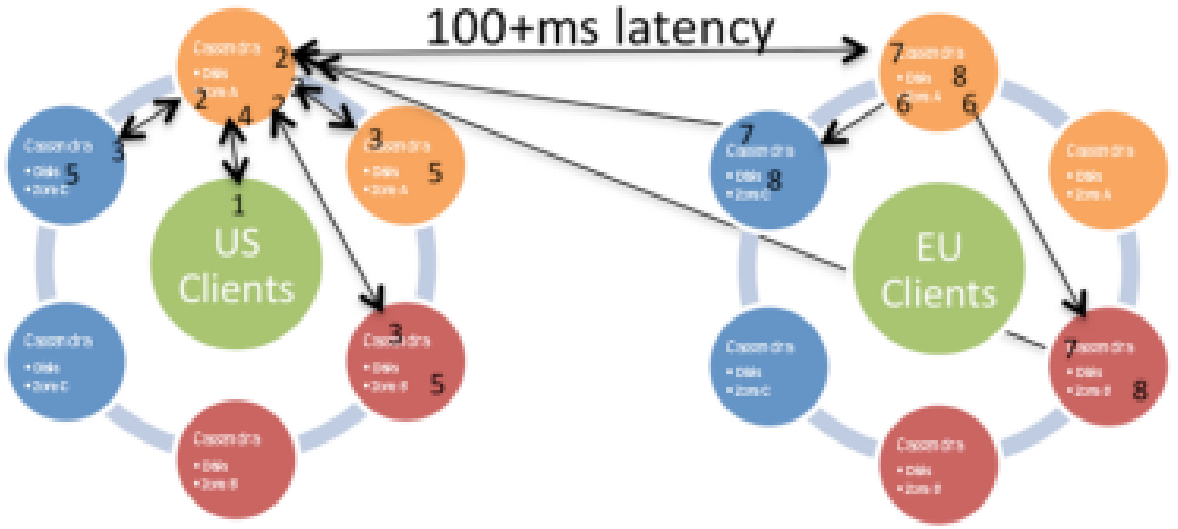
SSTable disk writes and compactions occur asynchronously

Data Flows for Multi-Region Writes

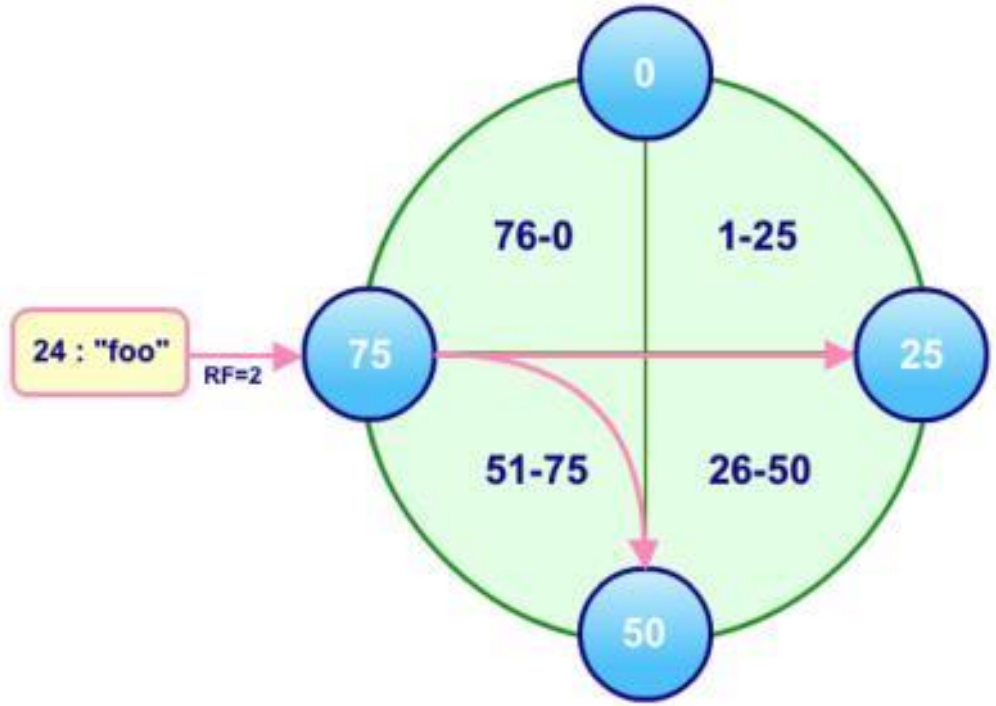
Consistency Level = Local Quorum

1. Client Writes to any Cassandra Node
2. Coordinator node replicates to other nodes Zones and regions
3. Local write acks returned to coordinator
4. Client gets ack when 2 of 3 local nodes are committed
5. Data written to internal commit log disks
6. When data arrives, remote node replicates data
7. Ack direct to source region coordinator
8. Remote copies written to commit log disks

If a node or region goes offline, hinted handoff completes the write when the node comes back up. Nightly global compare and repair jobs ensure everything stays consistent.



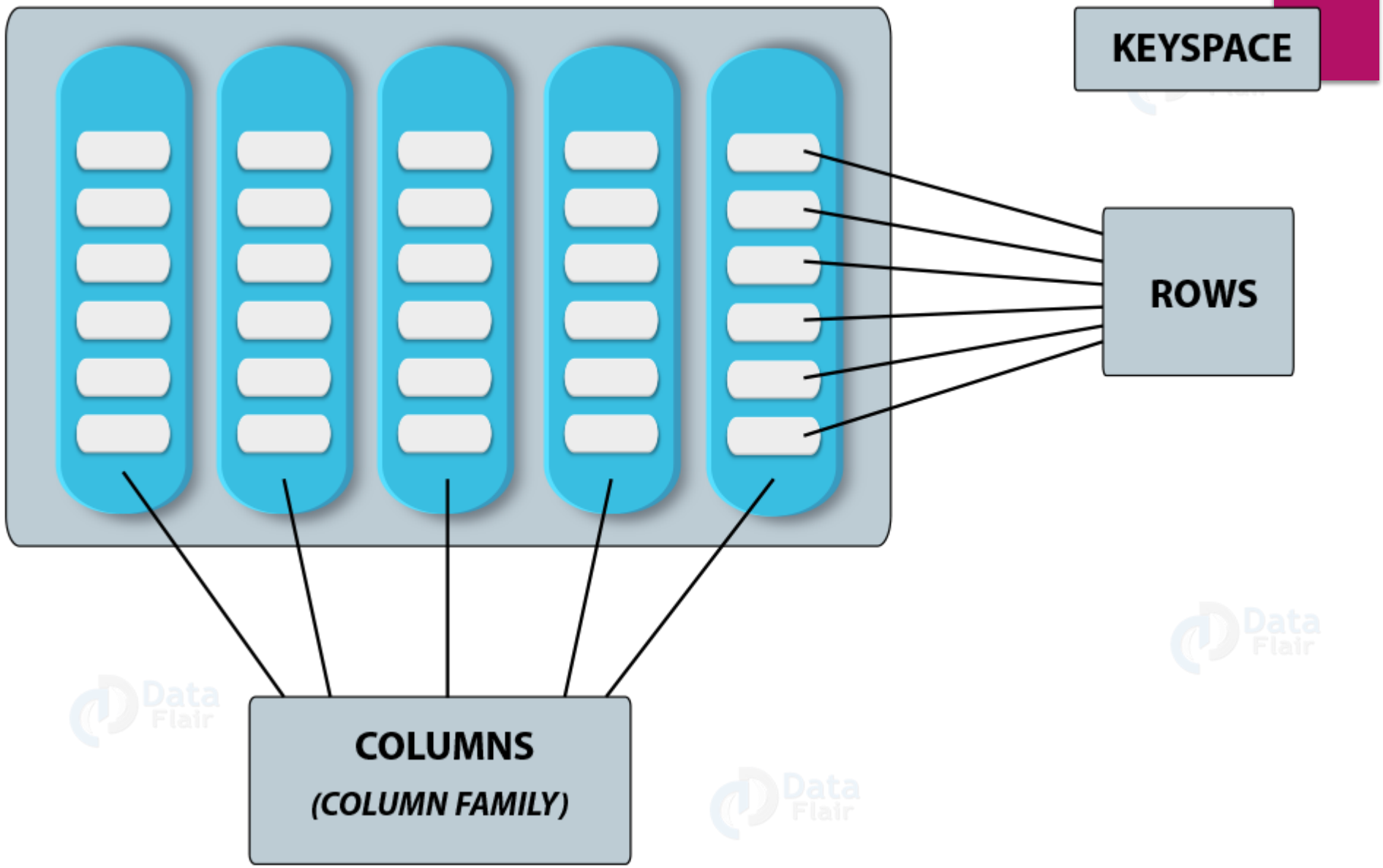
Cassandra Read Path



Data read command sent to closest replica - snitch

Digest commands sent to other replicas – CL

Read Repair Chance 10% - digest all replicas


KEYSPACE
ROWS
COLUMNS
(COLUMN FAMILY)

Relational *VS* NoSQL Database

Attribute	Relational DB	NoSQL DB
Data Modeling	Perceptive	Descriptive
ACID Transaction	Full Support	Not Always with Full Support
ETL	Required	May not be Required
Scalability	Not Scalable	Scalable by Design
High Availability	Local Cluster	Designed for Distributed Environment
Secondary Index	Full Support	Partial Support
Security	High	Not at Granular Level
SQL Support	Full Support	Not Always with SQL support
Sharding	Forced to Do when Scaling	Native
Multiple Data Center Replication	Limited Support	Mostly Supported
Asynchronous Query	Limited Support	Supported