# *Notes on Cassandra*

**Cassandra is a NoSQL database**

- Stores data in a key:value pair format within a wide-row, column-family structure
- Cassandra is NOT Relational
- Cassandra does not store data in tables
- Cassandra does not use the SQL query language, but uses a very similar CQL, Cassandra query language

# *Notes on Cassandra*

**Cassandra is**

- Massively scalable, free, open source
  - Like Mongo, the community edition is free
  - Enterprise C* users can purchase support and add-on features through a vendor like DataStax
- Designed for High Performance and High Scalability
- Designed for High Availability, fault tolerant with no SPOF
- Abbreviated as C*

**Cassandra's Heritage. Based on:**

- Google Big Table which is the Core foundation for many Google services and is the foundation for Cassandra's internal storage model

- Amazon Dynamo: Which supports many of Amazon's core services and is the foundation for Cassandra's distributed backbone

- Facebook -- which developed and open-sourced Cassandra

**Cassandra**

Cassandra provides the benefits of these technologies

- But, It improves them for Cassandra's needs
- C* Data model is a partitioned row store (with roots in BigTable)
- Peer-to-peer distributed architecture (roots in Dynamo)

**Cassandra Concepts**

Although not really stored in a "table", data in Cassandra is

- Row-oriented: Each row is an aggregate with column families representing meaningful, related chunks of data within that aggregate.

- Column-oriented: Each column family defines a record with sets of related data. You then think of a row as a collection of related records in all column families.

## Cassandra Terminology

- **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 allow addition of new columns to tables with no downtime. Tables contain partitions, which contain columns.

## Cassandra Terminology

- **Partition**: defines the mandatory part of the primary key all rows in Cassandra must have. All queries should 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 column clustering keys.

- **Column**: A single piece of data with a type which belongs to a row.

**Cassandra Concepts**

- Cassandra uses "CQL" – Cassandra Query Language
- Allows users a familiar row-column-based, SQL-like language
- BUT – no joins

# *Notes on Cassandra*

C* is a peer-to-peer, fully distributed system where

- All nodes are equal (no masters)
- Data is partitioned (replicated & sharded) among multiple nodes in a cluster
- Sharding and replication are configurable
- No node is a single point of failure (SPOF)
- Any node may be read from or written to

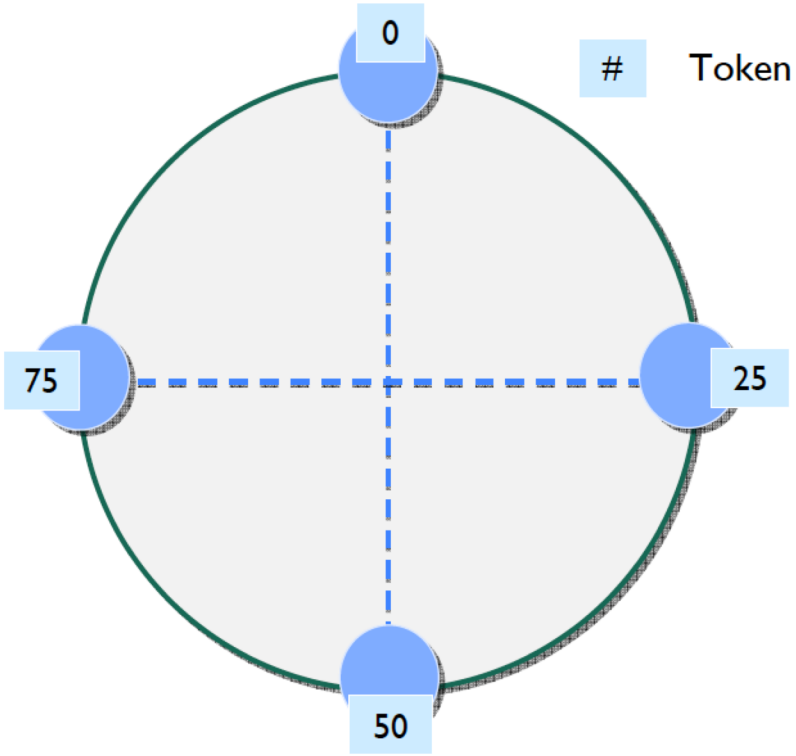
# *Notes on Cassandra*

C* "Instance" = A collection of independent nodes running C*

- configured into a cluster
- All nodes are peers (i.e. they all serve the same function)
- A node joins a cluster based on its configuration
- Data is distributed across all nodes in a cluster (1)
- All nodes perform the same function
- Nodes store data and service client compute requests
- A client may read/write to any node, which becomes the coordinator for servicing that particular request
- Nodes can have different capacity/resources available (e.g. memory, CPU, disk)
- C* distributes load based on the available resources

C* "Cluster" = A "ring" of C* nodes

- Each node is identified by a token
- Data is partitioned across the nodes by token
- Each node has responsibility for a subset of data



# *Notes on Cassandra*

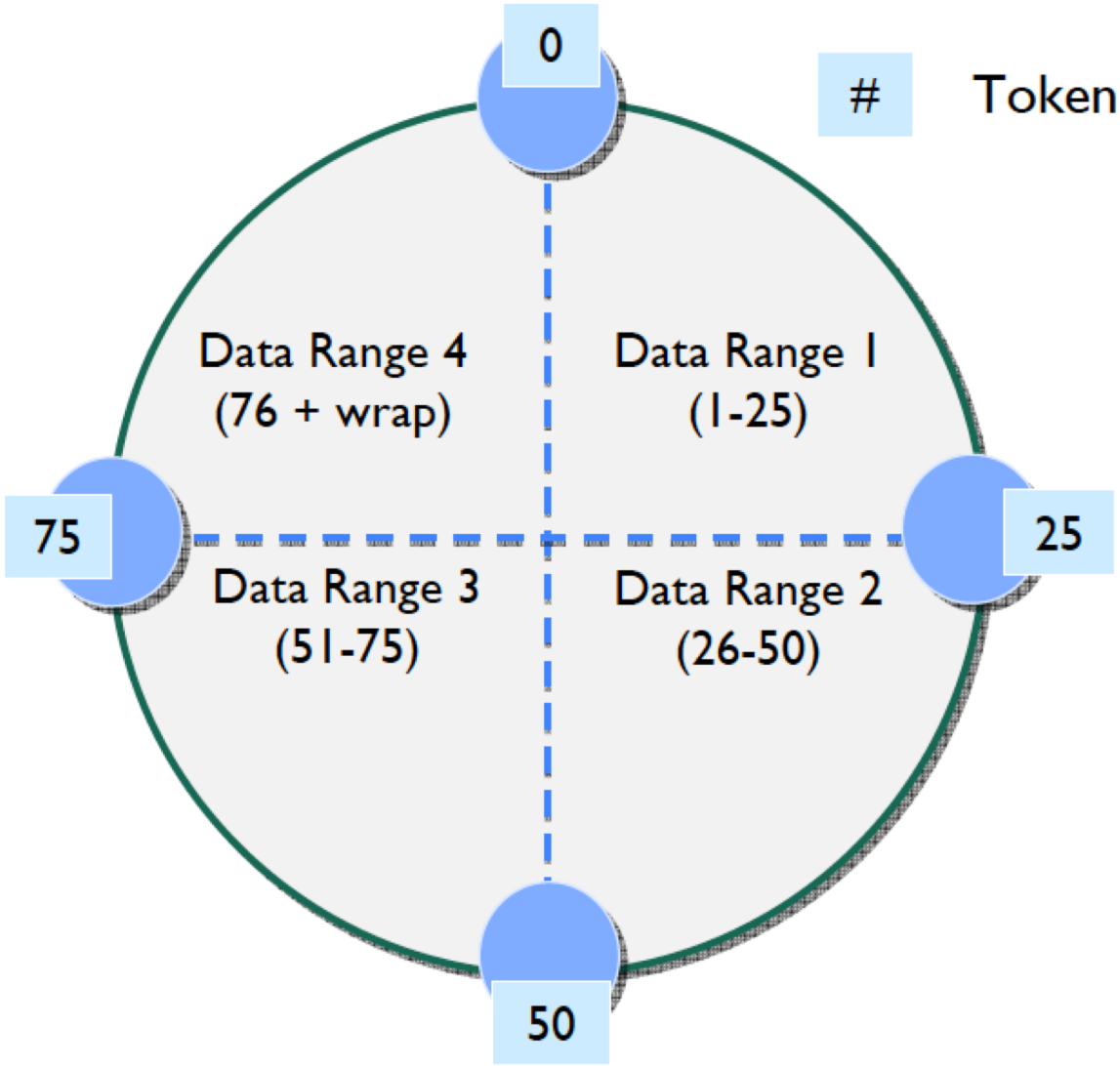
- Each node is assigned a token (a number), whose value determines
  - The logical position of the node in the ring
  - The range of data the node is assigned

- A partition key (also called **row key**) uniquely identifies data stored in Cassandra
  - The partition key is extracted from the data being stored
  - It is part of the data's primary key
  - The partition key is hashed, and the hash is used to partition the data across the cluster based on the range of data for each node

# *Notes on Cassandra*

- Partitioning distributes the data randomly around the ring
- So each node shares the load equally – which is critical for scaling

- **This [overly simplistic] cluster has 4 nodes with tokens of 0, 25, 50, 75**
  - Each node holds data with partition keys that fall between its token and the next token
    - e.g. The node with token=25 is responsible for data with a partition key hash of 12
  - The last node is the predecessor of the first (forming the ring)


0
#
Token
Data Range 4
(76 + wrap)
Data Range 1
(1-25)
75
25
Data Range 3
(51-75)
Data Range 2
(26-50)
50

Let's look at an example

- Consider the Stock Trade table that tracks stock trades
  - Its primary key is ((stock_symbol, trade_date), trade_id)
  - The partition key is the first element of the primary key
  - In this case a combination of stock_symbol and trade_date
- So ALL trades for a stock on a given day reside on the same node
  - The trade_id is a UUID - a universally unique identifier
  - It is a surrogate key – similar to the relational concept of a sequentially assigned auto-incremented identifier

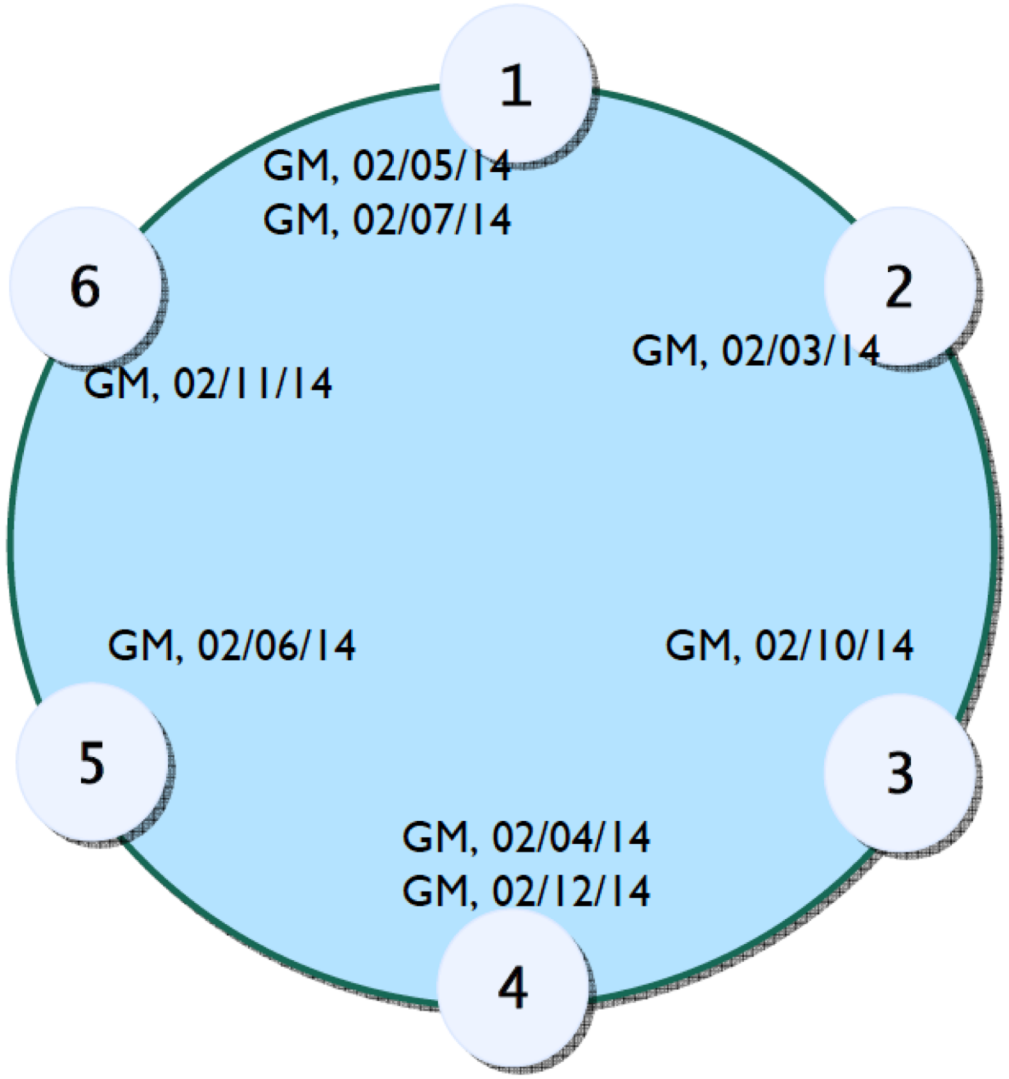
Let's look at an example

- CQL:

```
CREATE TABLE Trade (
      stock_symbol VARCHAR,
      trade_date TIMESTAMP,
      trade_id TIMEUUID,
      description VARCHAR,  // etc.
PRIMARY KEY ((stock_symbol, trade_date), trade_id)
);
```

Let's look at an example

- Let's assume we have trades for GM, on dates Feb. 3-12, 2014
- Assume that the partition mechanism results in the data distribution below
- Note how the trades for a given day are on one node
- There can be many trades for a given day - all on one node
- Different days are distributed across different nodes
- You can easily add more nodes to handle more data
- New trades will be distributed across the new nodes


1
GM, 02/05/14
GM, 02/07/14
2
GM, 02/03/14
3
GM, 02/10/14
4
GM, 02/04/14
GM, 02/12/14
5
GM, 02/06/14
6
GM, 02/11/14

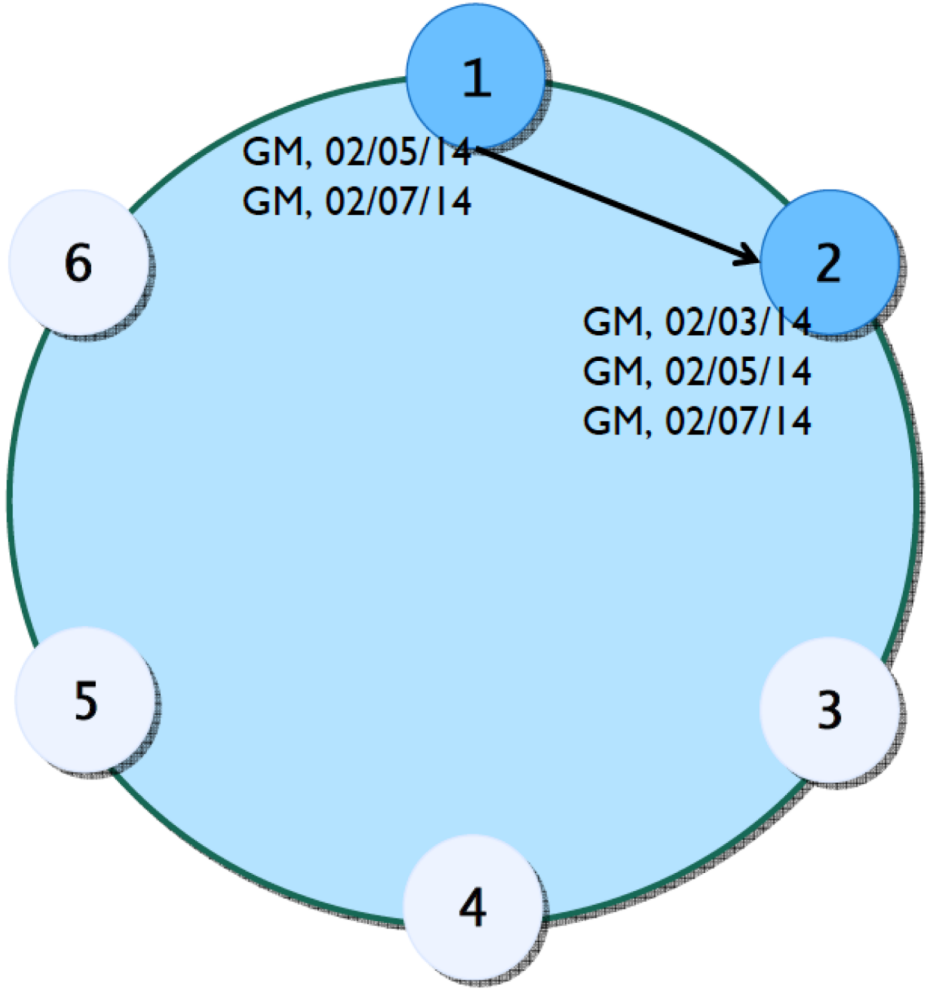
**Configurable Replication**

C* replicates data on multiple nodes for high availability and scalability

- All replicas are equal - no primary or master replica
- Replication defined at the data level via the replication factor "RF" (total number of replicas)
  - RF 1: One copy on a single node
  - RF 2: Two copies on two different nodes, etc.
- Provides very flexible replication

Example Below: This is a 6-node cluster with RF=2 showing data on two nodes

– In this simple example, the data is replicated on nodes 1 and 2
– So node 2 holds its own data, plus the data replicated from 1



Side Note: The CAP Theorem

Three core systemic requirements for distributed systems

– **Consistency**: All nodes see the latest data

– **Availability**: Requests get non-error responses

– **Partition Tolerance**: The system can lose messages and keep working (i.e. tolerant to network partition failures)

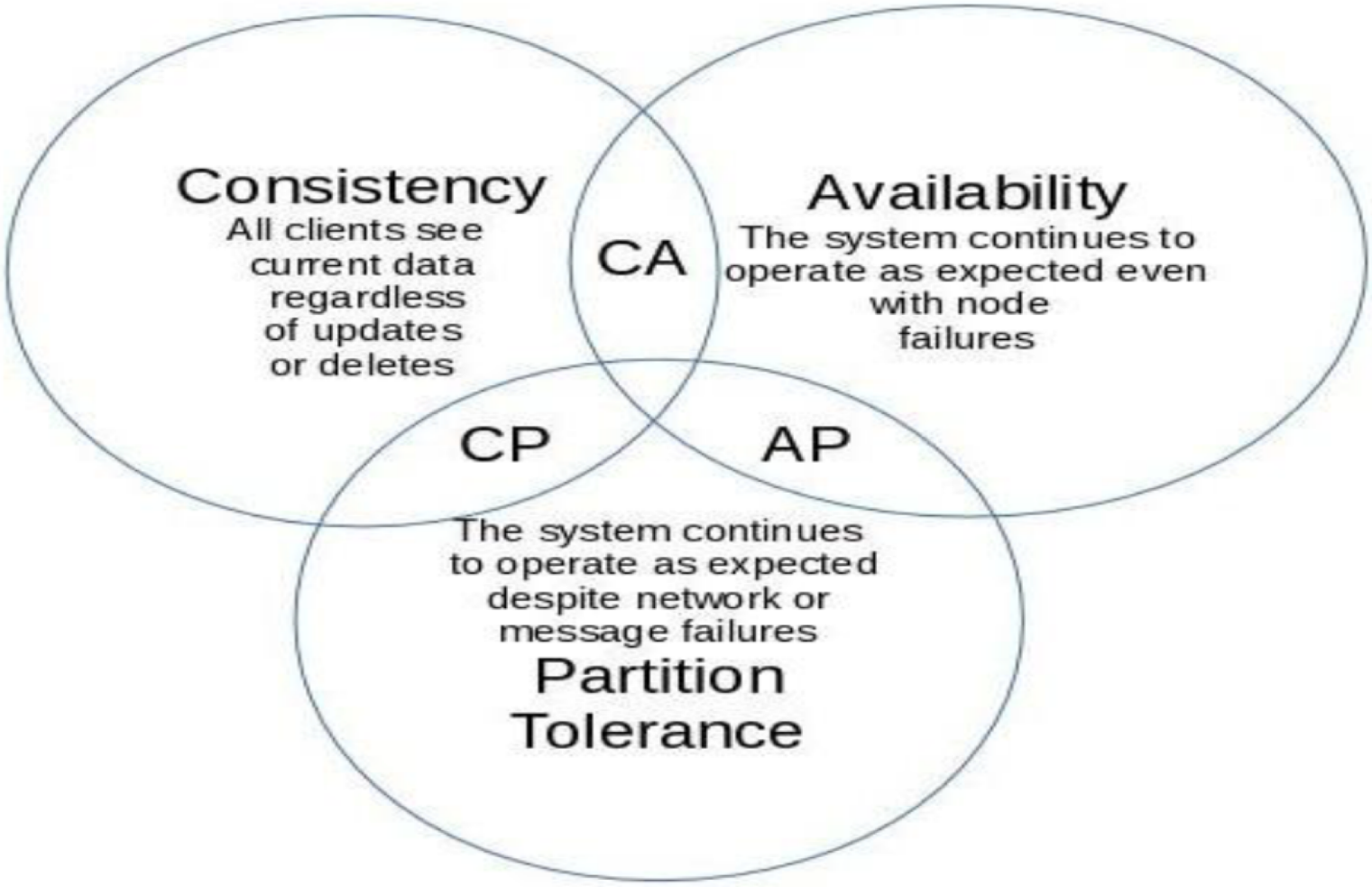
**CAP Theorem**: It is Impossible for a distributed system to guarantee all three of the above

- It illustrates the tradeoffs when building a system that may suffer partition failures - which is any distributed system

Consider your choices when 2 datacenters can't talk (i.e. a partition failure)

- **Sacrifice availability** and stop serving requests
- **Sacrifice consistency** and continue serving requests

**Bottom Line**: The data distributed across two datacenters can become inconsistent


Consistency
All clients see current data regardless of updates or deletes
CA
Availability
The system continues to operate as expected even with node failures
CP
AP
The system continues to operate as expected despite network or message failures
Partition Tolerance

**CAP Theorem**:

Consistency, Availability, Partition Tolerance

- You can't have all three.
- Pick TWO.

Trade-Offs:

A Relational/ACID system may maximize **consistency**

At the expense of **availability** and/or **throughput**

NoSQL systems will maximize **availability**

At the expense of **consistency**

# *Notes on Cassandra*

For example, Cassandra, implements the concept of **Eventual Consistency**

- C* will allow writes to nodes in the datacenter it can reach
- When the network failure is fixed, and the partition is available, C* will bring all the nodes up to date
- However, until then, the data will be inconsistent

C* provides what is called **Tunable Consistency**

Data is replicated to multiple nodes

Consider what happens when an update occurs?

– The update may not propagate to all replicas immediately

What happens if a read occurs before an update is propagated to all replicas?

– The read may retrieve older (pre-update) data from a replica

– What about consistency?

# *Notes on Cassandra*

C* provides what is called **Tunable Consistency**

Tradeoff among Availability, Consistency, Partition Tolerance

– CAP Theorem shows how a distributed system can't guarantee all three at once

- Cassandra uses tunable consistency to balance these demands
- The default config maximizes Availability and Partition Tolerance
- One can increase consistency (as needed) at the expense of Availability and Partition Tolerance

# *Notes on Cassandra*

- When you do a **write** in Cassandra, the consistency level specifies on how many replicas the write must succeed before returning an acknowledgement to the client application.
- The following consistency levels (chart on next slide) are available, with ANY being the lowest consistency (but highest availability), and ALL being the highest consistency (but lowest availability).
- QUORUM is a good middle-ground ensuring strong consistency, yet still tolerating some level of failure.

| LEVEL | DESCRIPTION |
|---|---|
| ANY | A write must be written to at least one node. If all replica nodes for the given row key are down, the write can still succeed once a hinted handoff has been written. Note that if all replica nodes are down at write time, an ANY write will not be readable until the replica nodes for that row key have recovered. |
| ONE | A write must be written to the commit log and memory table of at least one replica node. |
| QUORUM | A write must be written to the commit log and memory table on a quorum of replica nodes. |
| LOCAL_QUORUM | Awrite must be written to the commit log and memory table on a quorum of replica nodes in the same data center as the coordinator node. Avoids latency of inter-data center communication. |
| EACH_QUORUM | A write must be written to the commit log and memory table on a quorum of replica nodes in a// data centers. |
| ALL | A write must be written to the commit log and memory table on all replica nodes in the cluster for that row key. |

A "hinted" handoff is an update that can't run now, but will eventually process wen the node is back up.

# *Notes on Cassandra*

- When you do a **read** in Cassandra, the consistency level specifies on how many replicas must respond before returning the result to the client application.
- The following read consistency levels are available, with ONE being the lowest consistency (but highest availability), and ALL being the highest consistency (but lowest availability).
- QUORUM is a good middle-ground ensuring strong read consistency, yet still tolerating some level of failure.

| | |
|---|---|
| ONE | Returns a response from the closest replica (as determined by the snitch). By default, a read repair runs in the background to make the other replicas consistent. |
| QUORUM | Returns the record with the most recent timestamp once a quorum of replicas has responded. |
| LOCAL_QUORUM | Returns the record with the most recent timestamp once a quorum of replicas in the current data center as the coordinator node has reported. Avoids latency of inter-data center communication. |
| EACH_QUORUM | Returns the record with the most recent timestamp once a quorum of replicas in each data center of the cluster has responded. |
| ALL | Returns the record with the most recent timestamp once all replicas have responded. The read operation will fail if a replica does not respond. |

The "snitch" is the C* software component that manages the network topology and routes client requests efficiently.
The "snitch" manages replication.

For more information:

Great Tutorial:  DataStax Academy

https://academy.datastax.com/paths

Great Tutorial:  TutorialsPoint

https://www.tutorialspoint.com/cassandra/index.htm

Great Overview Article:  Dzone

https://dzone.com/articles/an-introduction-to-apache-cassandra