Apache Cassandra - 360

Compiled by Navaneetha Babu C

Course Outline

- Introduction to NoSQL and Apache Cassandra
- Cassandra Low Level Architecture
- Cassandra Data Model - Intro
- Compound Primary Keys
- Advanced Capabilities
- Composite Partition Keys
- Indexes and Secondary Indexes
- Counters

Course Outline

- Collections
- Data Consistency and other Cassandra key Features
- Cassandra Administration Components

Course Outline

- Introduction to NoSQL and Apache Cassandra
- Cassandra Low Level Architecture
- Cassandra Data Model - Intro
- Compound Primary Keys
- Advanced Capabilities
- Composite Partition Keys
- Indexes and Secondary Indexes
- Counters
- Collections
- Data Consistency and other Cassandra key Features
- Cassandra Administration Components

Chapter 1

Introduction to NoSql and Apache Cassandra

Compiled by Navaneetha Babu C

Course Outline

- ❑ **Introduction to NoSQL and Apache Cassandra**
- ❑ Cassandra Low Level Architecture
- ❑ Cassandra Data Model - Intro
- ❑ Compound Primary Keys
- ❑ Advanced Capabilities
- ❑ Composite Partition Keys
- ❑ Indexes and Secondary Indexes
- ❑ Counters
- ❑ Collections
- ❑ Data Consistency and other Cassandra key Features
- ❑ Cassandra Administration Components

Introduction to NoSQL and Apache Cassandra

Big Data

Data are getting created without bounds :

- Financial Transactions
- Sensor Networks
- Server Logs
- e-Mails and Text Messages
- Social Media
- Machine Feeds

And we are generating data faster than ever

- Automation
- Faster Internet Connectivity
- User-Generated Contents
- IoT
- Bots

Introduction to NoSQL and Apache Cassandra

Big data

Twitter processes more than 500+ million tweets per day

Facebook users generate 7 billion comments and “Likes”

YouTube gets 30 Million users per day

5 Billion videos are watched in Youtube a day

Every Minute Amazon makes $283,000 sales

Google gets 3.5 Million searches every minute

400,000 new tweets every minute

Is Your Big Data Hot or Cold?

A multi-temperature data management solution refers in part to having

- Data that are frequently accessed on fast storage which is referred as hot data
- Data that are rarely accessed data stored on the slowest storage an organization has are considered as cold data
- It's been common to use temperature terminology, specifically a range from cold to hot, to describe the levels of tiered service available to data storage customers.
- The levels have been differentiated according to how crucial to current business the stored data is and how frequently it will be accessed

Is Your Big Data Hot or Cold?

- These terms likely originated according to where the data was historically stored: hot data was close to the heat of the spinning drives and the CPUs, and cold data was on tape or a drive far away from the data center floor

- There are no standard industry definitions of what hot and cold mean when applied to data storage, so you'll find them used in different ways, which makes comparing services challenging.

- Generally, the hot data requires the fastest and most expensive storage because it's accessed more frequently, and cold data that is accessed less frequently can be stored on slower, and consequently, less expensive media.

Hot Data

- Hot storage is data that needs to be accessed right away. If the stored information is business-critical and you can't wait for it when you need it, that's a candidate for hot storage.
- The hotter the service, the more likely that it will use the latest drives, fastest transport protocols, and be located near to the client or in multiple regions as needed.
- Data stored in the hottest tier might use solid-state drives, which are optimized for lower latency and higher transactional rates compared to traditional hard drives.
- No matter the storage media used, the workloads in hot data storage require fast and consistent response times.
- Hot storage services also are tailored for workloads with many small transactions, such as capturing telemetry data, messaging, and data transformation.

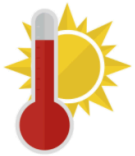
Cold Data

- Cold data is data that are accessed less frequently and also doesn't require the fast access. That includes data that is no longer in active use and might not be needed for months, years, decades, or maybe ever.

- Data retrieval and response time for cold cloud storage systems are typically much slower than services designed for active data manipulation.

- Cold data is usually stored on lower performing and less expensive storage environments in-house or in the cloud.

Introduction to NoSQL and Apache Cassandra

	Cold	Hot
		
Access Speed	Slow	Fast
Access Frequency	Seldom or Never	Frequent
Data Volume	Low	High
Storage Media	Slower drives, LTO, offline	Faster drives, durable drives, SSDs
Cost	Lower	Higher

Common Data Stores - Relational Database

- A relational database, or RDB, is a database which uses a relational model of data.
- Data is organized into tables. Each table has a schema which defines the columns for that table. The rows of the table, which each represent an actual record of information, must conform to the schema by having a value (or a NULL value) for each column
- Each row in the table has its own unique key, also called a primary key. Typically this is an integer column called “ID.” A row in another table might reference this table’s ID, thus creating a relationship between the two tables. When a column in one table references the primary key of another table, we call this a foreign key.
- Using this concept of primary keys and foreign keys, we can represent incredibly complex data relationships using incredibly simple foundations.
- SQL, which stands for structured query language, is the industry standard language for interacting with relational databases.
- SQL enforces Schema and ACID Properties.

When to Use Relational Database

- Use a database for storing your business critical information. Databases are the most durable and reliable type of data store. Anything that you need to store permanently should go in a database.

- Relational databases are typically the most mature databases: they have withstood the test of time and continue to be an industry standard tool for the reliable storage of important data.

- It's possible that your data doesn't conform nicely to a relational schema or your schema is changing so frequently that the rigid structure of a relational database is slowing down your development. In this case, you can consider using a non-relational database instead.

Introduction to NoSQL and Apache Cassandra

Common Data Stores – NoSql – Non-Relational DataBase

- These non-relational databases are often called “NoSQL Databases”.
- They have roughly the same characteristics as SQL databases (durable, resilient, persistent, replicated, distributed, and performant) except for the major difference of **not enforcing schemas** (or enforcing only very loose schemas).
- NoSQL databases can be categorized into a few types, but there are two primary types which come to mind when we think of NoSQL databases: document stores and wide column stores.

Common Data Stores – NoSql – Non-Relational DataBase

Document Store

- A document store is basically a fancy key-value store where the key is often omitted and never used
- The values are blobs of semi-structured data, such as JSON or XML, and we treat the data store like it's just a big array of these blobs.
- The query language of the document store will then allow you to filter or sort based on the content inside of those document blobs.
- The popular Document based data store is MongoDB

Common Data Stores – NoSql – Non-Relational DataBase

Wide Column Store

- A wide column store is somewhere in between a document store and a relational DB.
- It still uses tables, rows, and columns like a relational DB, but the names and formats of the columns can be different for various rows in the same table.
- This strategy combines the strict table structure of a relational database with the flexible content of a document store.
- Popular wide column stores you may have heard of are **Cassandra** and BigTable.

When to Use NoSQL Database

- Non-relational databases are most suited to handling **large volumes** of data and/or **unstructured data**.
- They're extremely popular in the world of Big Data because writes are fast.
- NoSQL databases don't enforce complicated cross-table schemas, so writes are unlikely to be a bottleneck in a system using NoSQL.
- Non-relational databases offer a lot of flexibility to developers, so they are also popular with early-stage startups

Common Data Stores – Key-Value Store

- Another way to store non-relational data is in a key-value store.
- A key-value store is basically a production-scale hashmap: a map from keys to values. There are no fancy schemas or relationships between data. No tables or other logical groups of data of the same type. Just keys and values, that's it.
- Both Redis and Memcached are in-memory key-value stores
- Since they are in-memory, they support configurable eviction policies. We will eventually run out of memory for storing keys and values, so we'll need to delete some.
- The most popular strategies are Least Recently Used (LRU) and Least Frequently Used (LFU). These eviction policies make key-value stores an easy and natural way to implement a cache.

When to Use Key-value Database

- Key-value stores are good for simple applications that need to store simple objects temporarily

Introduction to NoSQL and Apache Cassandra

NoSQL

- Stands for **N**ot **O**nly **SQL**

- Key Features are
  - Non-relational
  - Don’t Require Schema
  - Data are Replicated to Multiple Nodes(identical, fault-tolerant0
  - Data can be Partitioned
  - No Single Point of Failure
  - Horizontally Scalable
  - Cheap and Easy to Implement(Open Source)
  - Massive Write Performance
  - Fast Access

NoSQL

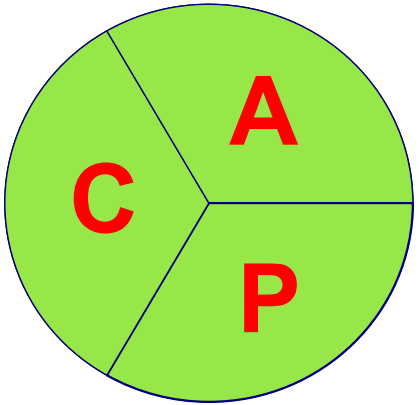
Major Disadvantages are

- Doesn't Fully support Relational Features
  - No Join, Group By, Order By(Except within Partitions
  - No Referential Integrity constraints across the Partitions
- No Declarative languages except few
- Relaxed ACID – NoSQL supports CAP Theorem

CAP Theorem

Three properties of a distributed system (sharing data)

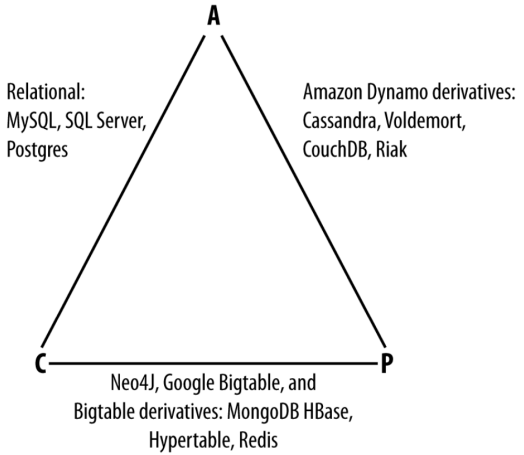
- **Consistency:**
  All database clients will read the same value for the same query, even given concurrent updates.
- **Availability:**
  All database clients will always be able to read and write data.
- **Partition-tolerance:**
  The database can be split into multiple machines; it can continue functioning in the face of network segmentation breaks.



CAP Theorem

A consistency model determines rules for visibility and apparent order of updates
Example:

- Row X is replicated on nodes M and N
- Client A writes row X to node N
- Some period of time t elapses
- Client B reads row X from node M



Does client B see the write from client A?

- Consistency is a continuum with tradeoffs

For NOSQL, the answer would be: "maybe"

CAP theorem states: *"strong consistency can't be achieved at the same time as availability and partition-tolerance"*

Introduction to NoSQL and Apache Cassandra

Categories of NoSQL

1. Key-value
   Example: DynamoDB, Voldermort

2. Document-based
   Example: MongoDB, CouchDB

3. Column-based
   Example: BigTable, Cassandra, Hbase

4. Graph-based
   Example: Neo4J, InfoGrid

“No-schema” is a common characteristics of most NOSQL storage systems
Provide “flexible” data types

Apache Cassandra

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

Cassandra bases its distribution design on Amazon's Dynamo and its data model on Google's Bigtable, with a query language similar to SQL. Created at Facebook, it now powers cloud-scale applications across many industries.

Apache Cassandra

- The Cassandra data store is an open source Apache project. Cassandra originated at Facebook in 2007 to solve its inbox search problem. The company had to deal with large volumes of data in a way that was difficult to scale with traditional methods. Specifically, the team had requirements to handle huge volumes of data in the form of message copies, reverse indices of messages, and many random reads and many simultaneous random writes.

- The team was led by Jeff Hammerbacher, with Avinash Lakshman, Karthik Ranganathan, and Facebook engineer on the Search Team Prashant Malik as key engineers. The code was released as an open source Google Code project in July 2008. During its tenure as a Google Code project in 2008, the code was updatable only by Facebook engineers, and little community was built around it as a result.

Introduction to NoSQL and Apache Cassandra

Apache Cassandra

- In March 2009, it was moved to an Apache Incubator project, and on February 17, 2010, it was voted into a top-level project. The committers, many of whom have been with the project since 2010/2011, represent companies, including Twitter, LinkedIn, and Apple, as well as independent developers.

- As commercial interest in Cassandra grew, the need for production support became apparent. Jonathan Ellis, the first Apache Project Chair for Cassandra, and his collea- gue Matt Pfeil formed a services company called DataStax

- During the period from 2010 to 2016, the Apache Project matured Cassandra over a series of releases from 0.6 to 3.0. While the original API provided by Cassandra was based on Apache Thrift, the introduction of the Cassandra Query Language in the 0.8 release marked a major shift toward improved usability and developer productivity due to its similarity with SQL.

- After the 3.0 release in 2016, Nate McCall took on the role of Apache Project Chair for Cassandra. This period has been marked by continued growth in the community, with enterprises including Apple, Facebook/Instagram, Netflix, and Uber providing increased contributions to the project.

Introduction to NoSQL and Apache Cassandra

Apache Cassandra

Distributed and Decentralized

- Cassandra is distributed, which means that it is capable of running on multiple machines while appearing to users as a unified cluster

- Much of its design and code base are specifically engineered toward not only making it work across many different machines, but also for optimizing performance across multiple data center racks, and even for a single Cassandra cluster running across geographically dispersed data centers. You can confidently write data to anywhere in the cluster and Cassandra will get it.

- Once you start to scale many other data stores (MySQL, Bigtable), some nodes need to be set up as primary replicas in order to organize other nodes, which are set up as secondary replicas. Cassandra, however, is decentralized, meaning that every node is identical; no Cassandra node performs certain organizing operations distinct from any other node. Instead, Cassandra features a peer-to-peer architecture and uses a gossip protocol to maintain and keep in sync a list of nodes that are alive or dead.

Apache Cassandra

Distributed and Decentralized

- The fact that Cassandra is decentralized means that there is no single point of failure. All of the nodes in a Cassandra cluster function exactly the same. This is sometimes referred to as “server symmetry.” Because they are all doing the same thing.

- In many distributed data solutions (such as RDBMS clusters), you set up multiple copies of data on different servers in a process called replication, which copies the data to multiple machines so that they can all serve simultaneous requests and improve performance. Typically this process is not decentralized, as in Cassandra, but is rather performed by defining a primary/secondary relationship.

- Decentralization, therefore, has two key advantages: it’s simpler to use than primary/secondary, and it helps you avoid outages. It is simpler to operate and maintain a decentralized store than a primary/secondary store because all nodes are the same. That means that you don’t need any special knowledge to scale; setting up 50 nodes isn’t much different from setting up one. There’s next to no configuration required to support it. Because all of the replicas in Cassandra are identical, failures of a node won’t disrupt service.

Apache Cassandra

Elastic Scalability

- Scalability is an architectural feature of a system that can continue serving a greater number of requests with little degradation in performance.
- Vertical scaling : simply adding more processing capacity and memory to your existing machine. It is the easiest way to achieve this.
- Horizontal scaling means adding more machines that have all or some of the data on them so that no one machine has to bear the entire burden of serving requests. But then the software itself must have an internal mechanism for keeping its data in sync with the other nodes in the cluster.
- Elastic scalability refers to a special property of horizontal scalability. It means that your cluster can seamlessly scale up and scale back down. To do this, the cluster must be able to accept new nodes that can begin participating by getting a copy of some or all of the data and start serving new user requests without major disruption or recon- figuration of the entire cluster.

Apache Cassandra

Elastic Scalability

- No Restart Requires while scaling in and scaling down.
- We don’t have to manually rebalance the data yourself. Just add another machine. Cassandra will find it and start sending it work.

Apache Cassandra

High Availability and Fault Tolerance

- Cassandra is highly available. You can replace failed nodes in the cluster with no downtime, and you can replicate data to multiple data centers to offer improved local performance and prevent downtime if one data center experiences a catastrophe such as fire or flood.

Apache Cassandra

Tuneable Consistency

- Consistency is an overloaded term in the database world, but for our purposes we will use the definition that a read always returns the most recently written value.

- Scenario

Consider the case of two customers attempting to put the same item into their shopping carts on an ecommerce site. If I place the last item in stock into my cart an instant after you do, you should get the item added to your cart, and I should be informed that the item is no longer available for purchase. This is guaranteed to happen when the state of a write is consistent among all nodes that have that data.

Apache Cassandra

Tuneable Consistency

- Cassandra is frequently called “eventually consistent,” which is a bit misleading. Out of the box, Cassandra trades some consistency in order to achieve total availability. But Cassandra is more accurately termed “tuneably consistent,” which means it allows you to easily decide the level of consistency you require, in balance with the level of availability

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

- The consistency level is a setting that clients must specify on every operation and that allows you to decide how many replicas in the cluster must acknowledge a write operation or respond to a read operation in order to be considered successful. That’s the part where Cassandra has pushed the decision for determining consistency out to the client.

- So if you like, you could set the consistency level to a number equal to the replication factor, and gain stronger consistency at the cost of synchronous blocking operations that wait for all nodes to be updated and declare success before returning

Apache Cassandra

Row Oriented

- Cassandra's data model can be described as a partitioned row store, in which data is stored in sparse multidimensional hashtables.
- "Sparse" means that for any given row you can have one or more columns, but each row doesn't need to have all the same columns as other rows like it (as in a relational model).
- "Partitioned" means that each row has a unique partition key used to distribute the rows across multiple data stores. Somewhat confusingly, this type of data model is also frequently referred to as a wide column store.
- In the relational storage model, all of the columns for a table are defined beforehand and space is allocated for each column whether it is populated or not. In contrast, Cassandra stores data in a multidimensional, sorted hash table. As data is stored in each column, it is stored as a separate entry in the hash table. Column values are stored according to a consistent sort order, omitting columns that are not populated, which enables more efficient storage and query processing.

Apache Cassandra

High Performance

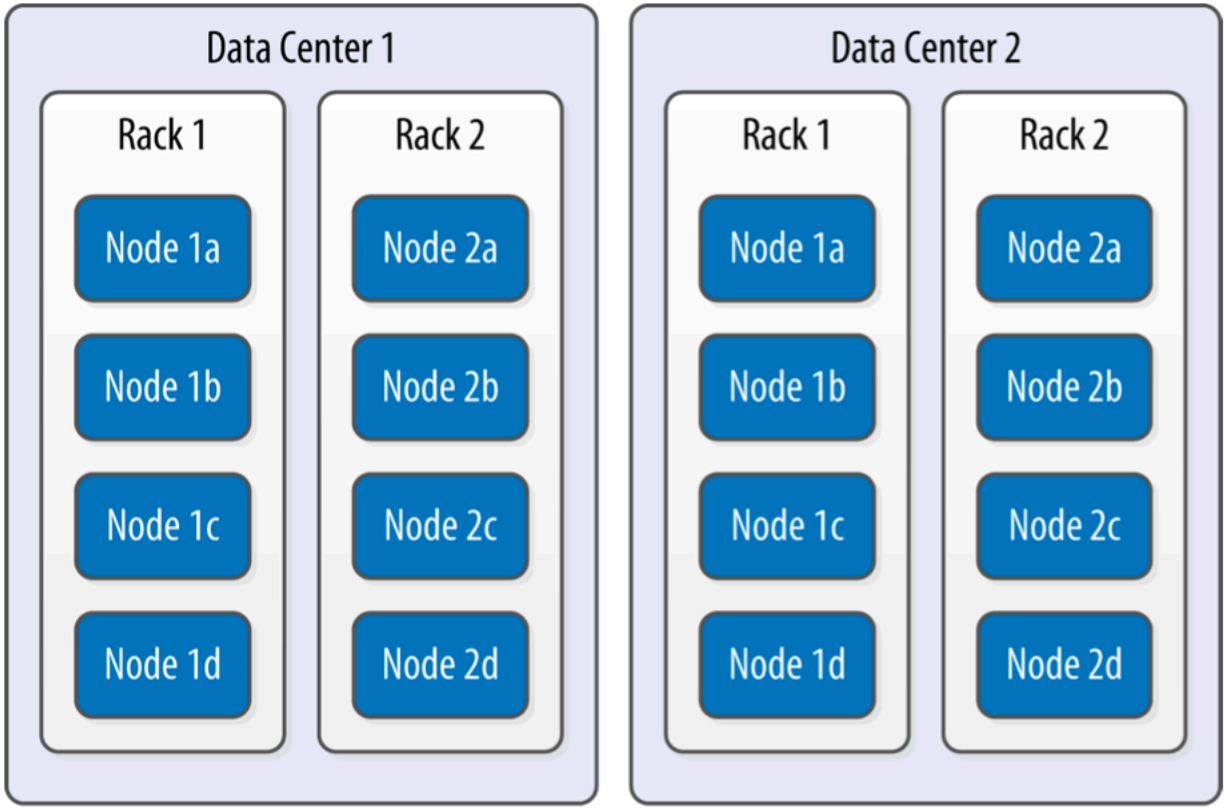
- Cassandra was designed specifically from the ground up to take full advantage of multiprocessor/multicore machines, and to run across many dozens of these machines housed in multiple data centers.
- It scales consistently and seamlessly to hundreds of terabytes.
- Cassandra has been shown to perform exceptionally well under heavy load. It consistently can show very fast throughput for writes per second on basic commodity computers, whether physical hardware or virtual machines.
- As you add more servers, you can maintain all of Cassandra's desirable properties without sacrificing performance.

Course Outline

- ❑ **Introduction to NoSQL and Apache Cassandra**
- ❑ **Cassandra Low Level Architecture**
- ❑ Cassandra Data Model - Intro
- ❑ Compound Primary Keys
- ❑ Advanced Capabilities
- ❑ Composite Partition Keys
- ❑ Indexes and Secondary Indexes
- ❑ Counters
- ❑ Collections
- ❑ Data Consistency and other Cassandra key Features
- ❑ Cassandra Administration Components

Cassandra Low Level Architecture

Racks and Cluster



Cassandra Low Level Architecture

Racks and Cluster

- Cassandra is frequently used in systems spanning physically separate locations.
- Cassandra provides two levels of grouping that are used to describe the topology of a cluster: data center and rack.
- A rack is a logical set of nodes in close proximity to each other, perhaps on physical machines in a single rack of equipment.
- A data center is a logical set of racks, perhaps located in the same building and connected by reliable network.
- Cassandra leverages the information you provide about your cluster's topology to determine where to store data, and how to route queries efficiently.
- Cassandra stores copies of your data in the data centers you request to maximize availability and partition tolerance, while preferring to route queries to nodes in the local data center to maximize performance.

Compiled by Navaneetha Babu C

Cassandra Low Level Architecture

Gossip Protocol and Failure Detection

- To support decentralization and partition tolerance, Cassandra uses a gossip protocol that allows each node to keep track of state information about the other nodes in the cluster. The gossiper runs every second on a timer.

- Gossip protocols (sometimes called epidemic protocols) generally assume a faulty network, are commonly employed in very large, decentralized network systems, and are often used as an automatic mechanism for replication in distributed databases.

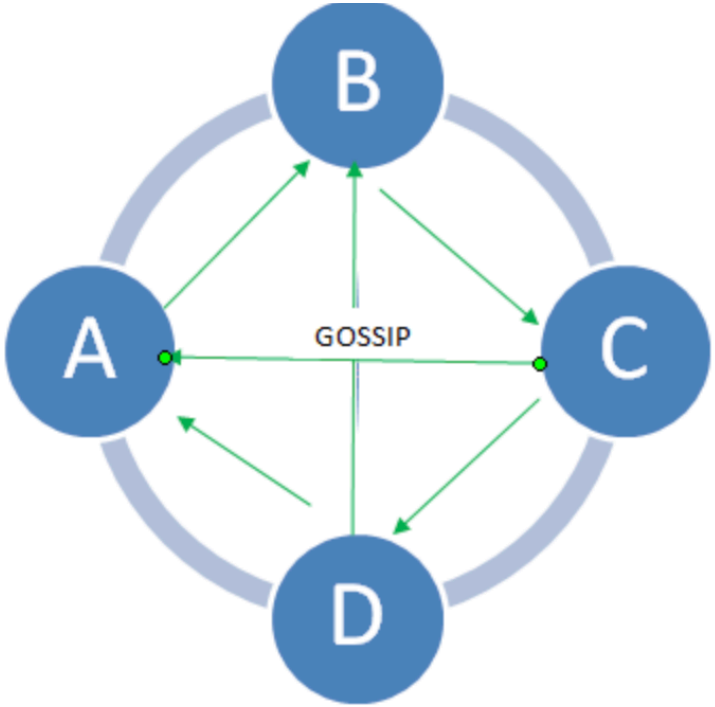
- They take their name from the concept of human gossip, a form of communication in which peers can choose with whom they want to exchange information.

- The gossip protocol in Cassandra is primarily implemented by the org.apache.cassandra.gms.Gossiper class, which is responsible for managing gossip for the local node. When a server node is started, it registers itself with the gossiper to receive endpoint state information.

- Because Cassandra gossip is used for failure detection, the Gossiper class maintains a list of nodes that are alive and dead.

Cassandra Low Level Architecture

Gossip Protocol and Failure Detection



PEER TO PEER DISTRIBUTION
MODEL OF CASSANDRA

Cassandra Low Level Architecture

Gossip Protocol and Failure Detection

Gossip is a peer-to-peer communication protocol in which nodes periodically exchange state information about themselves and about other nodes they know about.

How Gossiper Works?

- Once per second, the gossiper will choose a random node in the cluster and initialize a gossip session with it. Each round of gossip requires three messages.
- The gossip initiator sends its chosen friend a GossipDigestSyn message.
- When the friend receives this message, it returns a GossipDigestAck message.
- When the initiator receives the ack message from the friend, it sends the friend a GossipDigestAck2 message to complete the round of gossip.
- Gossip Protocols used TCP Communication.
- When the gossiper determines that another endpoint is dead, it “convicts” that endpoint by marking it as dead in its local list and logging that fact

Cassandra Low Level Architecture

Gossip Protocol and Failure Detection

- Cassandra has robust support for failure detection, as specified by a popular algorithm for distributed computing called Phi Accrual Failure Detector.

- Accrual failure detection is based on two primary ideas. The first general idea is that failure detection should be flexible, which is achieved by decoupling it from the application being monitored

- The second and more novel idea challenges the notion of traditional failure detectors, which are implemented by simple “heartbeats” and decide whether a node is dead or not dead based on whether a heartbeat is received or not.

- Therefore, the failure monitoring system outputs a continuous level of “suspicion” regarding how confident it is that a node has failed.

- This is desirable because it can take into account fluctuations in the network environment. For example, just because one connection gets caught up doesn’t necessarily mean that the whole node is dead. So suspicion offers a more fluid and proactive indication of the weaker or stronger possibility of failure based on interpretation (the sampling of heartbeats)

Cassandra Low Level Architecture

Snitches

- The job of a snitch is to provide information about your network topology so that Cassandra can efficiently route requests.

- The snitch will figure out where nodes are in relation to other nodes. The snitch will determine relative host proximity for each node in a cluster, which is used to determine which nodes to read and write from.

- As an example, let's examine how the snitch participates in a read operation. When Cassandra performs a read, it must contact a number of replicas determined by the consistency level. In order to support the maximum speed for reads, Cassandra selects a single replica to query for the full object, and asks additional replicas for hash values in order to ensure the latest version of the requested data is returned.

- The snitch helps to help identify the replica that will return the fastest, and this is the rep- lica which is queried for the full data.

- The default snitch (the SimpleSnitch) is topology unaware; that is, it does not know about the racks and data centers in a cluster, which makes it unsuitable for multiple data center deployments.

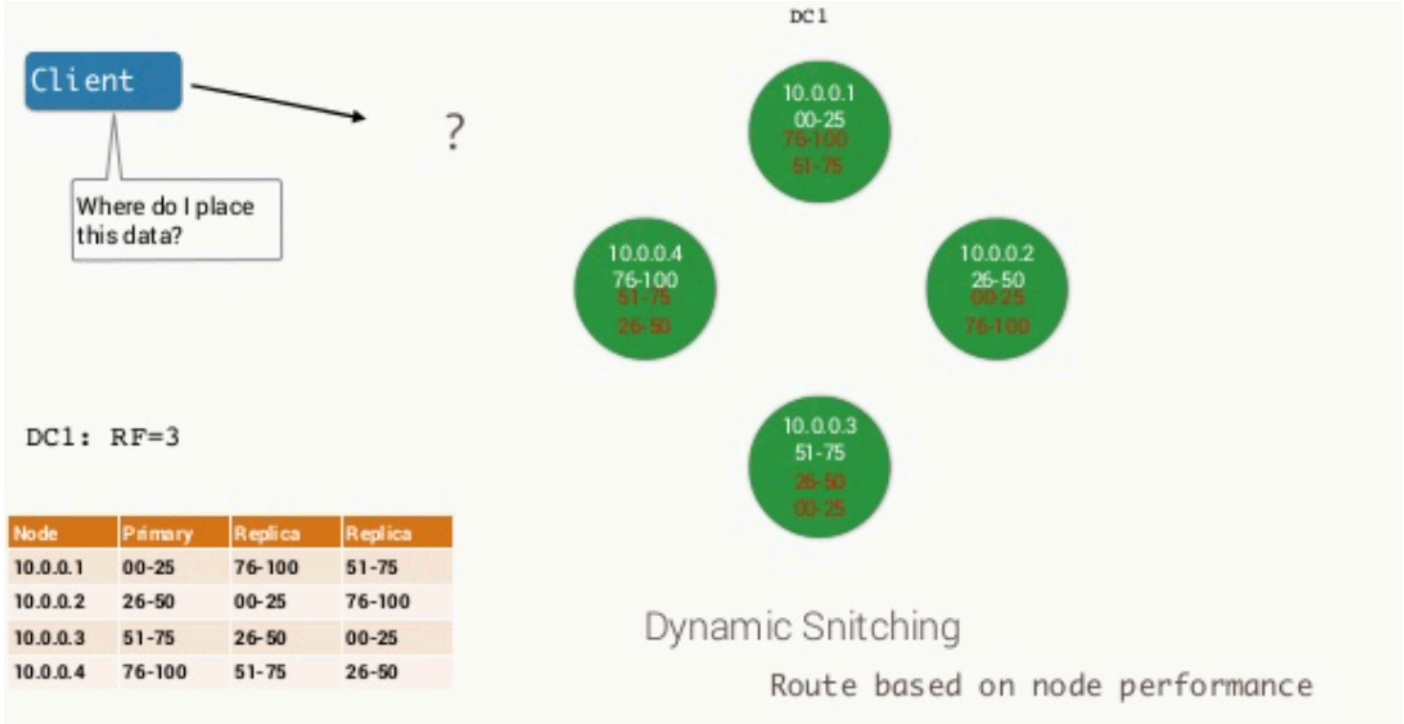
Cassandra Low Level Architecture

Snitches

- Cassandra comes with several snitches for different network topologies and cloud environments, including Amazon EC2, Google Cloud, and Apache Cloudstack.
- The snitches can be found in the package org.apache.cassandra.locator.
- While Cassandra provides a pluggable way to statically describe your cluster's topol- ogy, it also provides a feature called dynamic snitching that helps optimize the routing of reads and writes over time. Here's how it works.
- The selected snitch is wrapped with another snitch called the DynamicEndpointSnitch. The dynamic snitch gets its basic understanding of the topology from the selected snitch. It then monitors the performance of requests to the other nodes, even keeping track of things like which nodes are performing compaction.
- The performance data is used to select the best replica for each query. This enables Cassandra to avoid routing requests to replicas that are busy or performing poorly.
- The dynamic snitching implementation uses a modified version of the Phi failure detection mechanism used by gossip. The badness threshold is a configurable parameter that determines how much worse a preferred node must perform than the best- performing node in order to lose its preferential status.

Cassandra Low Level Architecture

Snitches



Compiled by Navaneetha Babu C

Cassandra Low Level Architecture

Snitches

Types of Snitches are

- Dynamic Snitch - Monitors the performance of reads from the various replicas and chooses the best replica based on this history.
- Simple Snitch - The SimpleSnitch is used only for single-datacenter deployments.
- RackInferring Snitch - Determines the location of nodes by rack and datacenter corresponding to the IP addresses.
- PropertyFile Snitch - Determines the location of nodes by rack and datacenter by a property file.
- GossipPropertyFile Snitch - Automatically updates all nodes using gossip when adding new nodes and is recommended for production.
- EC2 Snitch - Use the Ec2Snitch with Amazon EC2 in a single region.
- Ec2MultiRegion Snitch - Use the Ec2MultiRegionSnitch for deployments on Amazon EC2 where the cluster spans multiple regions.

Cassandra Low Level Architecture

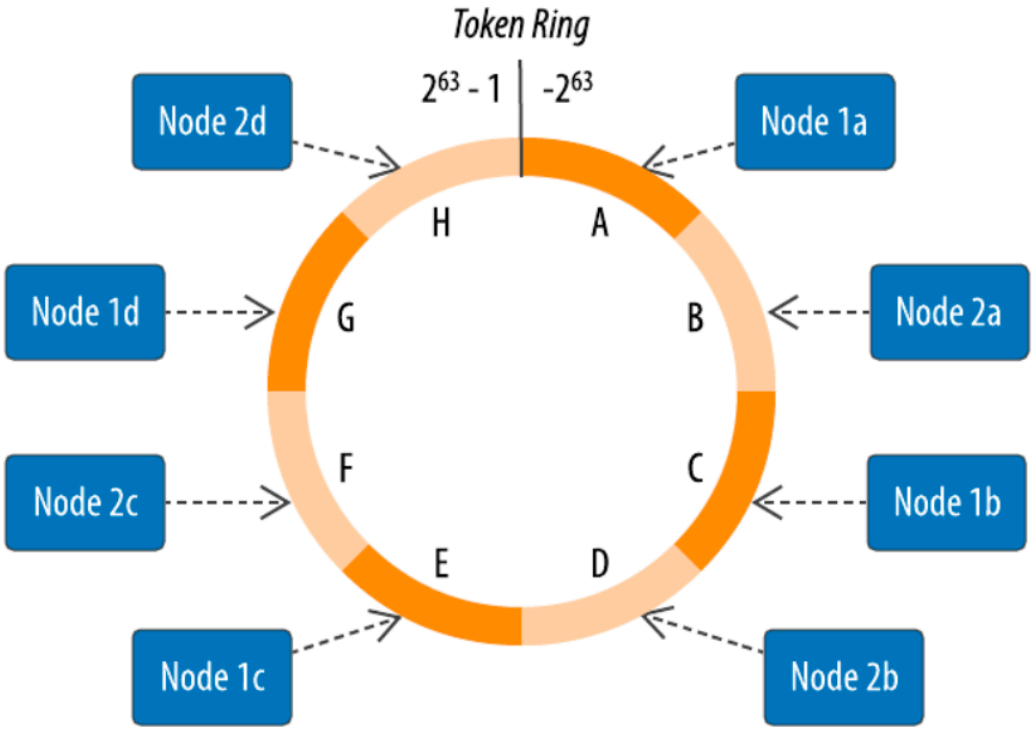
Rings and Tokens

Till now we have focussed on how Cassandra keeps track of the physical layout of nodes in a cluster. Let's shift gears and look at how Cassandra distributes data across these nodes.

- Cassandra represents the data managed by a cluster as a ring. Each node in the ring is assigned one or more ranges of data described by a token, which determines its position in the ring.
- In the default configuration, a token is a 64-bit integer ID used to identify each partition. This gives a possible range for tokens from −2^63 to 2^63−1.
- A node claims ownership of the range of values less than or equal to each token and greater than the last token of the previous node, known as a token range.
- The node with the lowest token owns the range less than or equal to its token and the range greater than the highest token, which is also known as the wrapping range.
- This particular arrangement is structured such that consecutive token ranges are spread across nodes in different racks.

Cassandra Low Level Architecture

Rings and Tokens



Cassandra Low Level Architecture

Rings and Tokens

- Data is assigned to nodes by using a hash function to calculate a token for the partition key. This partition key token is compared to the token values for the various nodes to identify the range, and therefore the node, that owns the data.

```
last_name | first_name | system.token(last_name)

 Rodriguez |       Mary |    -7199267019458681669
     Scott |     Isaiah |     1807799317863611380
    Nguyen |       Bill |     6000710198366804598
    Nguyen |      Wanda |     6000710198366804598

(5 rows)
```

- As you might expect, we see a different token for each partition, and the same token appears for the two rows represented by the partition key value “Nguyen.”

Cassandra Low Level Architecture

Virtual Nodes

- Early versions of Cassandra assigned a single token (and therefore by implication, a single token range) to each node, in a fairly static manner, requiring you to calculate tokens for each node. Although there are tools available to calculate tokens based on a given number of nodes, it was still a manual process to configure the initial_token property for each node in the cassandra.yaml file.

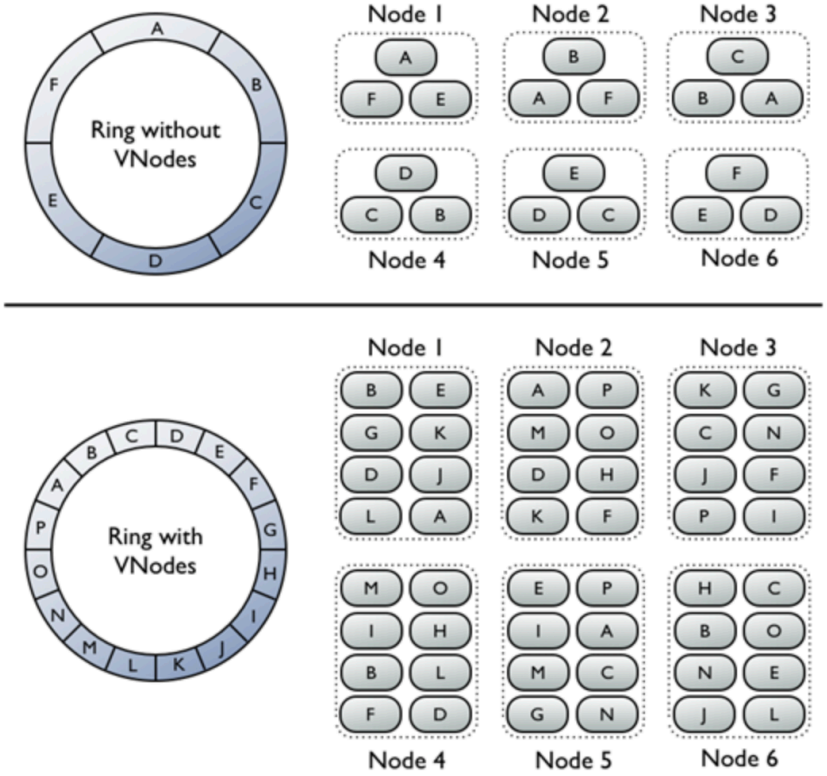
- This also made adding or replacing a node an expensive operation, as rebalancing the cluster required moving a lot of data.

- Cassandra's 1.2 release introduced the concept of virtual nodes, also called vnodes. Instead of assigning a single token to a node, the token range is broken up into multiple smaller ranges. Each physical node is then assigned multiple tokens. Historically, each node has been assigned 256 of these tokens, meaning that it represents 256 virtual nodes.

- Vnodes make it easier to maintain a cluster containing heterogeneous machines. For nodes in your cluster that have more computing resources available to them, you can increase the number of vnodes by setting the num_tokens property in the cassandra.yaml file. Conversely, you might set num_tokens lower to decrease the number of vnodes for less capable machines.

Cassandra Low Level Architecture

Virtual Nodes



Compiled by Navaneetha Babu C

Cassandra Low Level Architecture

Virtual Nodes

- Cassandra automatically handles the calculation of token ranges for each node in the cluster in proportion to their num_tokens value. Token assignments for vnodes are calculated by the org.apache.cassandra.dht.tokenallocator.ReplicationAware TokenAllocator class.

- A further advantage of virtual nodes is that they speed up some of the more heavy-weight Cassandra operations such as bootstrapping a new node, decommissioning a node, and repairing a node. This is because the load associated with operations on multiple smaller ranges is spread more evenly across the nodes in the cluster.

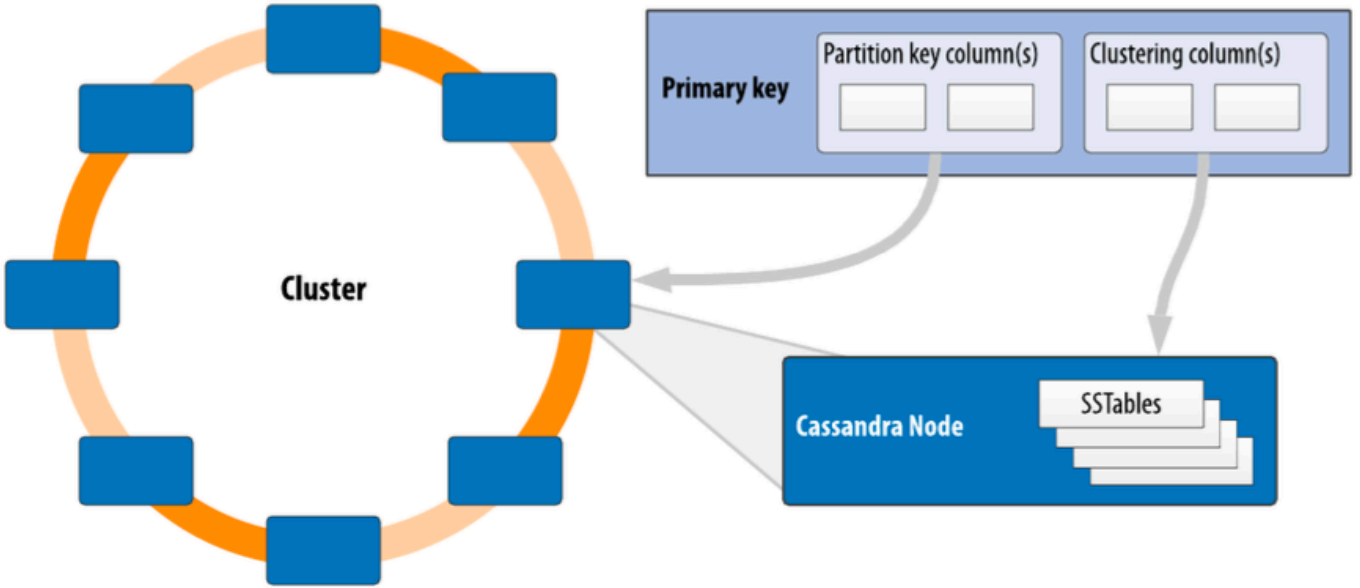
Cassandra Low Level Architecture

Partitioners

- A partitioner determines how data is distributed across the nodes in the cluster.
- Cassandra organizes rows in partitions. Each row has a partition key that is used to identify the partition to which it belongs.
- A partitioner, is a hash function for computing the token of a partition key. Each row of data is distributed within the ring according to the value of the partition key token.
- The role of the partitioner is to compute the token based on the partition key columns. Any clustering columns that may be present in the primary key are used to determine the ordering of rows within a given node that owns the token representing that partition.
- Cassandra provides several different partitioners in the org.apache.cassandra.dht package (DHT stands for *distributed hash table*).
- The Murmur3Partitioner was added in 1.2 and has been the default partitioner since then; it is an efficient Java implementation on the murmur algorithm developed by Austin Appleby. It generates 64-bit hashes. The previous default was the RandomPartitioner.

Cassandra Low Level Architecture

Partitioners



Cassandra Low Level Architecture

Replication Strategies

- A node serves as a replica for different ranges of data. If one node goes down, other replicas can respond to queries for that range of data.
- Cassandra replicates data across nodes in a manner transparent to the user, and the replication factor is the number of nodes in your cluster that will receive copies (replicas) of the same data. If your replication factor is 3, then three nodes in the ring will have copies of each row.
- The first replica will always be the node that claims the range in which the token falls, but the remainder of the replicas are placed according to the replication strategy
- Cassandra provides two primary implementations of this interface : SimpleStrategy and NetworkTopologyStrategy.
- The SimpleStrategy places replicas at consecutive nodes around the ring, starting with the node indicated by the partitioner.

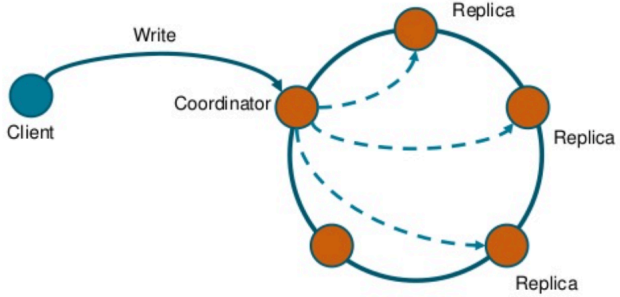
Cassandra Low Level Architecture

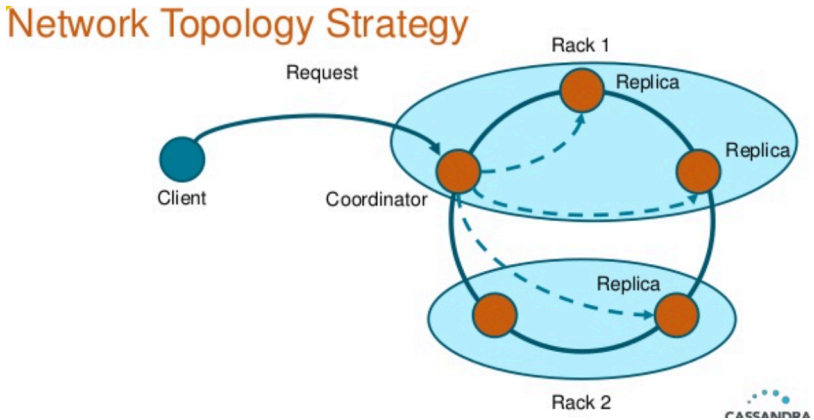
Replication Strategies

- The NetworkTopologyStrategy allows you to specify a different replication factor for each data center. Within a data center, it allocates replicas to different racks in order to maximize availability.

- The NetworkTopologyStrategy is recommended for keyspaces in production deployments, even those that are initially created with a single data center, since it is more straightforward to add an additional data center should the need arise.

Simple Strategy





Compiled by Navaneetha Babu C

Cassandra Low Level Architecture

Consistency Levels

- CAP theorem says that consistency, availability, and partition tolerance are traded off against one another.
- Cassandra provides tuneable consistency levels that allow you to make these trade-offs at a fine-grained level. You specify a consistency level on each read or write query that indicates how much consistency you require.
- A higher consistency level means that more nodes need to respond to a read or write query, giving you more assurance that the values present on each replica are the same.
- For read queries, the consistency level specifies how many replica nodes must respond to a read request before returning the data.
- For write operations, the consistency level specifies how many replica nodes must respond for the write to be reported as successful to the client. Because Cassandra is eventually consistent, updates to other replica nodes may continue in the background.

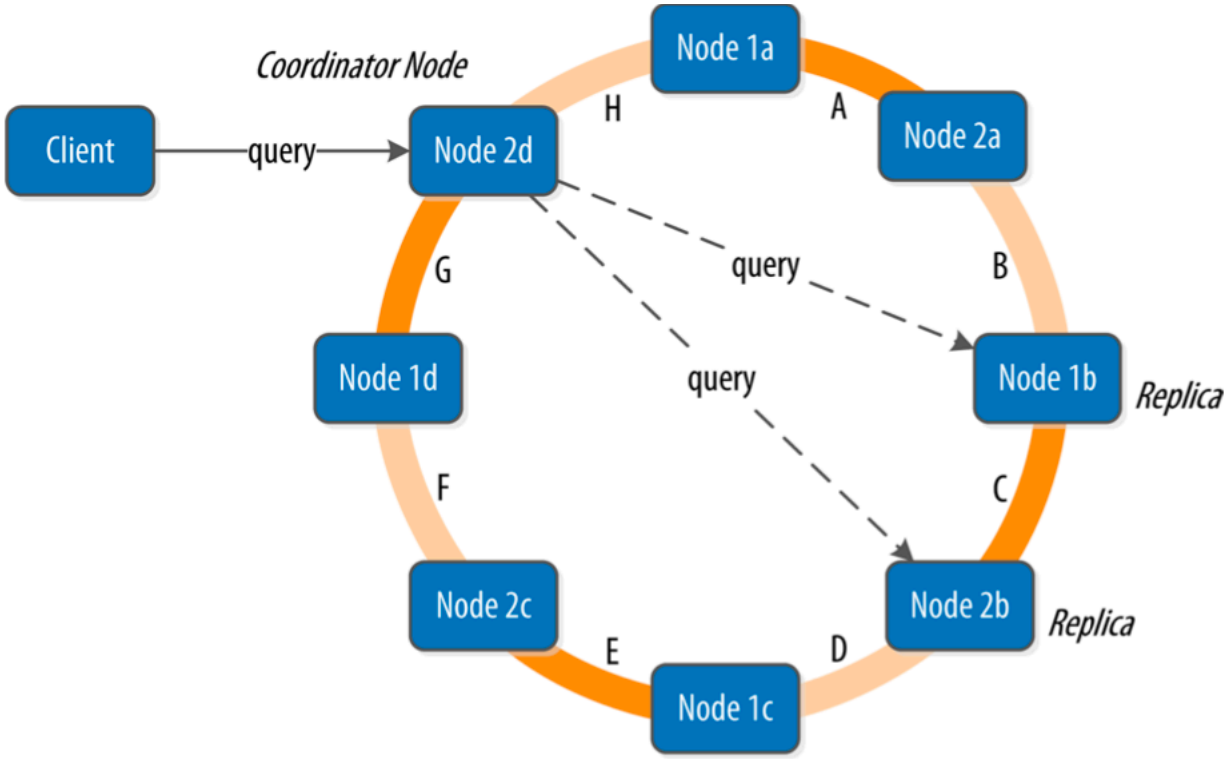
Cassandra Low Level Architecture

Consistency Levels

- The available consistency levels include ONE, TWO, and THREE, each of which specify an absolute number of replica nodes that must respond to a request.
- The QUORUM consistency level requires a response from a majority of the replica nodes.
- The ALL consistency level requires a response from all of the replicas.
- Consistency is tuneable in Cassandra because clients can specify the desired consistency level on both reads and writes.

Cassandra Low Level Architecture

Queries and Coordinator Nodes



Cassandra Low Level Architecture

Queries and Coordinator Nodes

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

Cassandra Low Level Architecture

Hinted Handoff

- Consider the following scenario: a write request is sent to Cassandra, but a replica node where the write properly belongs is not available due to network partition, hardware failure, or some other reason. In order to ensure general availability of the ring in such a situation, Cassandra implements a feature called hinted handoff.

- Hint is a Post-it Note that contains the information from the write request.

- If the replica node where the write belongs has failed, the coordinator will create a hint, which is a small reminder that says, “I have the write information that is intended for node B. I’m going to hang on to this write, and I’ll notice when node B comes back online; when it does, I’ll send it the write request.” That is, once it detects via gossip that node B is back online, node A will “hand off” to node B the “hint” regarding the write. Cassandra holds a separate hint for each partition that is to be written.

- This allows Cassandra to be always available for writes, and generally enables a clus- ter to sustain the same write load even when some of the nodes are down. It also reduces the time that a failed node will be inconsistent after it does come back online.

Cassandra Low Level Architecture

Hinted Handoff

- In general, hints do not count as writes for the purposes of consistency level. The exception is the consistency level ANY

- This consistency level means that a hinted handoff alone will count as sufficient toward the success of a write operation. That is, even if only a hint was able to be recorded, the write still counts as successful. Note that the write is considered durable, but the data may not be readable until the hint is delivered to the target replica.

- If a node is offline for some time, the hints can build up considerably on other nodes. Then, when the other nodes notice that the failed node has come back online, they tend to flood that node with requests, just at the moment it is most vulnerable

- Cassandra limits the storage of hints to a configurable time window. It is also possible to disable hinted handoff entirely.

Cassandra Low Level Architecture

Anti-Entropy

- Cassandra uses an anti-entropy protocol as an additional safeguard to ensure consistency.
- Anti-entropy protocols are a type of gossip protocol for repairing replicated data.
- They work by comparing replicas of data and reconciling differences observed between the replicas. Anti-entropy is used in Amazon's Dynamo, and Cassandra's implementation is modelled on that
- Replica synchronization is supported via two different modes known as read repair and anti-entropy repair
- Read repair refers to the synchronization of replicas as data is read. Cassandra reads data from multiple replicas in order to achieve the requested consistency level, and detects if any replicas have out-of-date values. If an insufficient number of nodes have the latest value, a read repair is performed immediately to update the out-of-date replicas. Otherwise, the repairs can be performed in the back-ground after the read returns.

Cassandra Low Level Architecture

Anti-Entropy

- Anti-entropy repair is a manually initiated operation performed on nodes as part of a regular maintenance process. This type of repair is executed by using a tool called nodetool.

- Running nodetool repair causes Cassandra to execute a validation compaction.

- During a validation compaction, the server initiates a TreeRequest/TreeReponse conversation to exchange Merkle trees with neighboring replicas. The Merkle tree is a hash representing the data in that table. If the trees from the different nodes don’t match, they have to be reconciled (or “repaired”) to determine the latest data values they should all be set to.

Cassandra Low Level Architecture

Lightweight Transactions and Paxos

- Imagine we are building a client that wants to manage user records as part of an account management application. In creating a new user account, we'd like to make sure that the user record doesn't already exist and to avoid unintentionally overwrite existing user data. So first we do a read to see if the record exists, and then only perform the create if the record doesn't exist.

- The behavior we're looking for is called linearizable consistency, meaning that we'd like to guarantee that no other client can come in between our read and write queries with their own modification. This is called as Linear Lightweight Transactions

- Cassandra's LWT implementation is based on Paxos. Paxos is a consensus algorithm that allows distributed peer nodes to agree on a proposal, without requiring a leader to coordinate a transaction. Paxos and other consensus algorithms emerged as alter- natives to traditional two-phase commit-based approaches to distributed transactions.

Cassandra Low Level Architecture

Lightweight Transactions and Paxos

- The basic Paxos algorithm consists of two stages: prepare/promise and propose/ accept. To modify data, a coordinator node can propose a new value to the replica nodes, taking on the role of leader.

- Other nodes may act as leaders simultaneously for other modifications. Each replica node checks the proposal, and if the proposal is the latest it has seen, it promises to not accept proposals associated with any prior proposals. Each replica node also returns the last proposal it received that is still in progress.

- If the proposal is approved by a majority of replicas, the leader commits the proposal, but with the caveat that it must first commit any in-progress proposals that preceded its own proposal.

- The Cassandra implementation extends the basic Paxos algorithm to support the desired read-before-write semantics. and to allow the state to be reset between transactions. It does this by inserting two additional phases into the algorithm,
    1. Prepare/Promise
    2. Read/Results
    3. Propose/Accept

- Thus, a successful transaction requires four round-trips between the coordinator node and replicas. This is more expensive than a regular write, which is why you should think carefully about your use case before using LWTs.

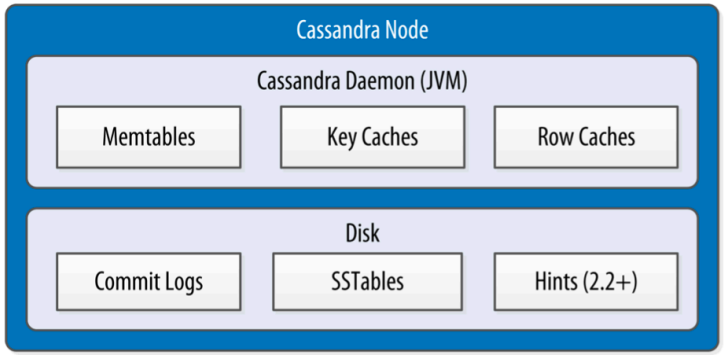
Cassandra Low Level Architecture

MemTables, SSTables and Commit Logs

- Cassandra stores data both in memory and on disk to provide both high performance and durability.
- When a node receives a write operation, it immediately writes the data to a commit log. The commit log is a crash-recovery mechanism that supports Cassandra's durability goals.
- A write will not count as successful on the node until it's written to the commit log, to ensure that if a write operation does not make it to the in-memory store
- it will still be possible to recover the data. If you shut down the node or it crashes unexpectedly, the commit log can ensure that data is not lost. That's because the next time you start the node, the commit log gets replayed.
- That's the only time the commit log is read. clients never read from it.

Cassandra Low Level Architecture

MemTables, SSTables and Commit Logs



- After it's written to the commit log, the value is written to a memory-resident data structure called the memtable.
- Each memtable contains data for a specific table.
- In early implementations of Cassandra, memtables were stored on the JVM heap, but improvements starting with the 2.1 release have moved some memtable data to native memory, with configuration options to specify the amount of on-heap and native memory available.

Cassandra Low Level Architecture

MemTables, SSTables and Commit Logs

- When the number of objects stored in the memtable reaches a threshold, the contents of the memtable are flushed to disk in a file called an SSTable(Sorted String Table)

- A new memtable is then created. This flushing is a nonblocking operation, multiple memtables may exist for a single table, one current and the rest waiting to be flushed. They typically should not have to wait very long, as the node should flush them very quickly unless it is overloaded.

- Each commit log maintains an internal bit flag to indicate whether it needs flushing. When a write operation is first received, it is written to the commit log and its bit flag is set to 1. There is only one bit flag per table, because only one commit log is ever being written to across the entire server.

- All writes to all tables will go into the same commit log, so the bit flag indicates whether a particular commit log contains anything that hasn't been flushed for a particular table. Once the memtable has been properly flushed to disk, the corresponding commit log's bit flag is set to 0, indicating that the commit log no longer has to maintain that data for durability purposes.

Cassandra Low Level Architecture

MemTables, SSTables and Commit Logs

- Once a memtable is flushed to disk as an SSTable, it is immutable and cannot be changed by the application. Despite the fact that SSTables are compacted, this compaction changes only their on-disk representation, it essentially performs the “merge” step of a mergesort into new files and removes the old files on success.

- Cassandra has supported the compression of SSTables in order to maximize use of the available storage. This compression is configurable per table.

- ll writes are sequential, which is the primary reason that writes perform so well in Cassandra. No reads or seeks of any kind are required for writing a value to Cassandra because all writes are append operations. This makes the speed of your disk one key limitation on performance.

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

Cassandra Low Level Architecture

Bloom Filters

- Bloom filters are used to boost the performance of reads.
- Bloom filters are very fast, nondeterministic algorithms for testing whether an element is a member of a set. They are nondeterministic because it is possible to get a false-positive read from a Bloom filter, but not a false-negative.
- Bloom filters work by mapping the values in a data set into a bit array and condensing a larger data set into a digest string using a hash function. The digest, by definition, uses a much smaller amount of memory than the original data would.
- The filters are stored in memory and are used to improve performance by reducing the need for disk access on key lookups. Disk access is typically much slower than memory access. So, in a way, a Bloom filter is a special kind of key cache.
- Cassandra maintains a Bloom filter for each SSTable. When a query is performed, the Bloom filter is checked first before accessing disk. Because false-negatives are not possible, if the filter indicates that the element does not exist in the set, it certainly doesn't; but if the filter thinks that the element is in the set, the disk is accessed to make sure.

Cassandra Low Level Architecture

Caching

As an additional mechanism to boost read performance, Cassandra provides three optional forms of caching:

- The key cache stores a map of partition keys to row index entries, facilitating faster read access into SSTables stored on disk. The key cache is stored on the JVM heap.
- The row cache caches entire rows and can greatly speed up read access for fre- quently accessed rows, at the cost of more memory usage. The row cache is stored in off-heap memory.
- The counter cache was added in the 2.1 release to improve counter performance by reducing lock contention for the most frequently accessed counters.

By default, key and counter caching are enabled, while row caching is disabled, as it requires more memory. Cassandra saves its caches to disk periodically in order to warm them up more quickly on a node restart.

Cassandra Low Level Architecture

Compaction

- SSTables are immutable, which helps Cassandra achieve such high write speeds. However, periodic compaction of these SSTables is important in order to support fast read performance and clean out stale data values.

- A compaction operation in Cassandra is performed in order to merge SSTables. During compaction, the data in SSTables is merged: the keys are merged, columns are combined, obsolete values are discarded, and a new index is created.

- Compaction is the process of freeing up space by merging large accumulated data- files. This is roughly analogous to rebuilding a table in the relational world. But the primary difference in Cassandra is that it is intended as a transparent operation that is amortized across the life of the server.

- On compaction, the merged data is sorted, a new index is created over the sorted data, and the freshly merged, sorted, and indexed data is written to a single new SSTable

Cassandra Low Level Architecture

Compaction

- Another important function of compaction is to improve performance by reducing the number of required seeks. There is a bounded number of SSTables to inspect to find the column data for a given key. If a key is frequently mutated, it's very likely that the mutations will all end up in flushed SSTables.

- Compacting them prevents the database from having to perform a seek to pull the data from each SSTable in order to locate the current value of each column requested in a read request.

- When compaction is performed, there is a temporary spike in disk I/O and the size of data on disk while old SSTables are read and new SSTables are being written.

- Cassandra supports multiple algorithms for compaction via the strategy pattern. The compaction strategy is an option that is set for each table.

  -- SizeTieredCompactionStrategy (STCS) is the default compaction strategy and is recommended for write-intensive tables
  -- LeveledCompactionStrategy (LCS) is recommended for read-intensive tables
  -- TimeWindowCompactionStrategy (TWCS) is intended for time series or other-wise date-based data.

Cassandra Low Level Architecture

Deletion and Tombstone

- A node could be down or unreachable when data is deleted, that node could miss a delete.
- When that node comes back online later and a repair occurs, the node could “resurrect” the data that had been previously deleted by re-sharing it with other nodes.
- To prevent deleted data from being reintroduced, Cassandra uses a concept called a *tombstone*.
- A tombstone is a marker that is kept to indicate data that has been deleted. When you execute a delete operation, the data is not immediately deleted. Instead, it’s treated as an update operation that places a tombstone on the value.
- A tombstone is similar to the idea of a “soft delete” from the relational world. Instead of actually executing a delete SQL statement, the application will issue an update statement that changes a value in a column called something like “deleted.” Programmers sometimes do this to support audit trails,
- Tombstones are not kept forever, instead they are removed as part of compaction.
- By default, it’s set to 864,000 seconds, the equivalent of 10 days. Cassandra keeps track of tombstone age, and once a tombstone is older than gc_grace_seconds, it will be garbage collected.

Course Outline

- **Introduction to NoSQL and Apache Cassandra**
- **Cassandra Low Level Architecture**
- Cassandra Data Model – Intro
- Compound Primary Keys
- Advanced Capabilities
- Composite Partition Keys
- Indexes and Secondary Indexes
- Counters
- Collections
- Data Consistency and other Cassandra key Features
- Cassandra Administration Components

Cassandra Data Model

KeySpace

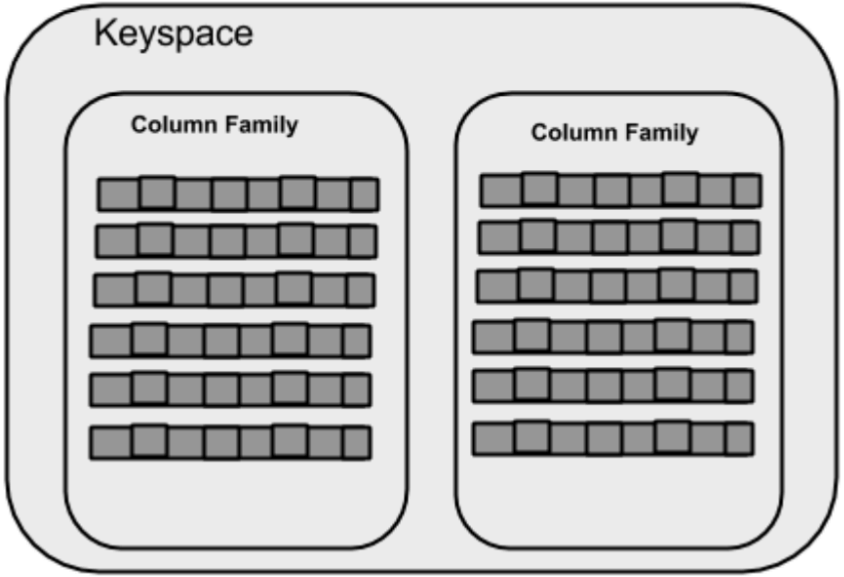
Keyspace is the outermost container for data in Cassandra. The basic attributes of a Keyspace in Cassandra are:

- Replication factor: It is the number of machines in the cluster that will receive copies of the same data.
- Replica placement strategy: It is nothing but the strategy to place replicas in the ring. We have strategies such as simple strategy (rack-aware strategy), old network topology strategy (rack-aware strategy), and network topology strategy (datacenter-shared strategy).
- Column families: Keyspace is a container for a list of one or more column families. A column family, in turn, is a container of a collection of rows. Each row contains ordered columns. Column families represent the structure of your data. Each keyspace has at least one and often many column families.

Cassandra Data Model

KeySpace

CREATE KEYSPACE Keyspace name WITH replication = {'class': 'SimpleStrategy', 'replication_factor' : 3};



Cassandra Data Model

Column Family

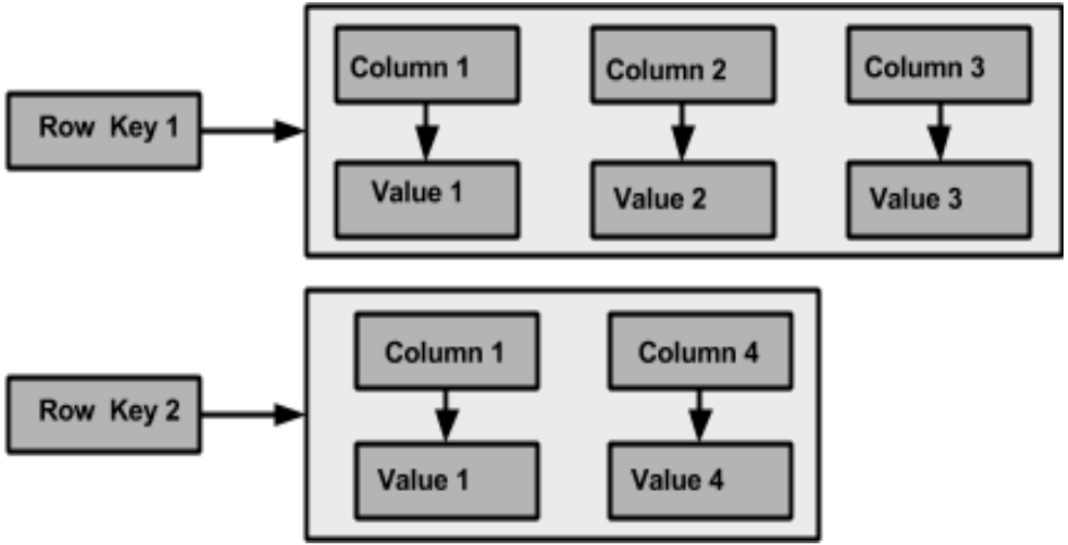
A column family is a container for an ordered collection of rows. Each row, in turn, is an ordered collection of columns. The following table lists the points that differentiate a column family from a table of relational databases.

Relational Table	Cassandra Column Family
A schema in a relational model is fixed. Once we define certain columns for a table, while inserting data, in every row all the columns must be filled at least with a null value.	In Cassandra, although the column families are defined, the columns are not. You can freely add any column to any column family at any time.
Relational tables define only columns and the user fills in the table with values.	In Cassandra, a table contains columns, or can be defined as a super column family.

Cassandra Data Model

Column Family

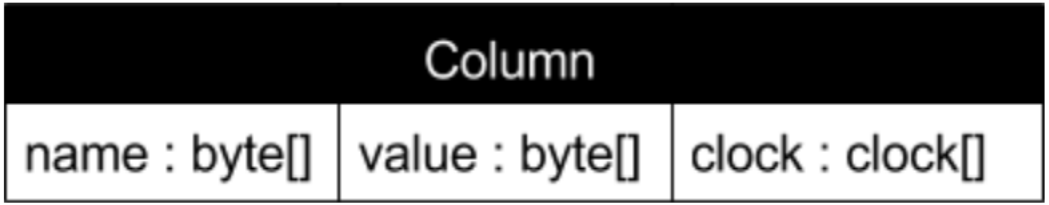
Unlike relational tables where a column family's schema is not fixed, Cassandra does not force individual rows to have all the columns.



Cassandra Data Model

Column

A column is the basic data structure of Cassandra with three values, namely key or column name, value, and a time stamp. Given below is the structure of a column.



Cassandra Data Model

CQL Commands

help – The HELP command displays a synopsis and a brief description of all cqlsh commands.

Copy – This command copies data to and from Cassandra to a file. Given below is an example to copy the table named emp to the file myfile.

```
COPY emp (emp_id, emp_city, emp_name, emp_phone, emp_sal) TO ‘myfile’;
```

DESCRIBE – This command describes the current cluster of Cassandra and its objects. The variants of this command are explained below.

Describe cluster: This command provides information about the cluster.

Describe Keyspaces: This command lists all the keyspaces in a cluster.

Describe tables: This command lists all the tables in a keyspace.

Cassandra Data Model

CQL Commands

SHOW – This command displays the details of current cqlsh session such as Cassandra version, host, or data type assumptions.

```
show host;
show version;
```

Cassandra Data Model

CQL Commands - Keyspace

- A keyspace in Cassandra is a namespace that defines data replication on nodes.
- A cluster contains one keyspace per node.
- Given below is the syntax for creating a keyspace using the statement CREATE KEYSPACE.
- Syntax: – CREATE KEYSPACE WITH – Example:

CREATE KEYSPACE “KeySpace Name” WITH replication = {'class': ‘Strategy name’, 'replication_factor' : ‘No.Of replicas’};

CQL Commands - Keyspace

- The replication option is to specify the Replica Placement strategy and the number of replicas wanted. The following table lists all the replica placement strategies.

Strategy name	Description
Simple Strategy'	Specifies a simple replication factor for the cluster.
Network Topology Strategy	Using this option, you can set the replication factor for each data-center independently.
Old Network Topology Strategy	This is a legacy replication strategy.

Cassandra Data Model

CQL Commands - Keyspace

- ALTER KEYSPACE can be used to alter properties such as the number of replicas and the durable_writes of a KeySpace.

- Syntax: – ALTER KEYSPACE WITH

- Example: ALTER KEYSPACE “KeySpace Name” WITH replication = {'class': ‘Strategy name’, 'replication_factor' : ‘No.Of replicas’};

Cassandra Data Model

CQL Commands - Keyspace

- You can drop a KeySpace using the command DROP KEYSPACE.
- Given below is the syntax for dropping a KeySpace.
- Syntax: – DROP KEYSPACE • Example: – DROP KEYSPACE NAVANEETH_KS;

Cassandra Data Model

CQL Commands – Create Table

- CREATE TABLE tablename(column1 name datatype PRIMARYKEY, column2 name data type, column3 name data type)

- Example: CREATE TABLE emp(emp_id int PRIMARY KEY, emp_name text, emp_city text, emp_sal varint, emp_phone varint);

Cassandra Data Model

CQL Commands – Alter Table

You can alter a table using the command ALTER TABLE. Using ALTER command, you can perform the following operations:

- Add a column
- Drop a column
- Update the options of a table using with keyword

Example:

- ALTER TABLE emp ADD emp_email text;
- ALTER TABLE emp DROP emp_email;

Cassandra Data Model

CQL Commands – Drop and Truncate Table

Drop table command:

- You can drop a table using the command Drop Table.
- Example: DROP TABLE emp;

Truncating a Table

- You can truncate a table using the TRUNCATE command.
- When you truncate a table, all the rows of the table are deleted permanently.

Example: TRUNCATE student;

Cassandra Data Model

CQL Commands – Insert Statements

```
INSERT INTO BANK_DETAILS(age, job, marital, education, default, balance, housing, loan, deposit)
VALUES (59, 'admin', 'married', 'ug', 'Yes', 2259, 'Yes', 'No', 'Yes');
```

Similar to SQL

Cassandra Data Model

CQL Commands – Updating a Table

- UPDATE is the command used to update data in a table.

- The following keywords are used while updating data in a table:
- Where: This clause is used to select the row to be updated.
- Set: Set the value using this keyword.
- Must: Includes all the columns composing the primary key.

- Example: UPDATE emp SET emp_city='Delhi', emp_sal=50000 WHERE emp_id=2;

Cassandra Data Model

CQL Commands – Deleting the Data

```
cqlsh:tushar> select * from emp ;

 emp_id | emp_city  | emp_name | emp_phone  | emp_sal
--------+-----------+----------+------------+---------
      1 | Hyderabad |      ram | 9848022338 |   50000
      2 |     Delhi |    robin | 9848022339 |   50000
      3 |   Chennai |   rahman | 9848022330 |    null

(3 rows)
cqlsh:tushar> DELETE FROM emp WHERE emp_id=3;
cqlsh:tushar> select * from emp ;

 emp_id | emp_city  | emp_name | emp_phone  | emp_sal
--------+-----------+----------+------------+---------
      1 | Hyderabad |      ram | 9848022338 |   50000
      2 |     Delhi |    robin | 9848022339 |   50000
```

Cassandra Data Model

CQL Commands – CQL Data Types

Data Type	Constants	Description
ascii	strings	Represents ASCII character string
bigint	integers	Represents 64-bit signed long
blob	blobs	Represents arbitrary bytes
Boolean	booleans	Represents true or false
counter	integers	Represents counter column
decimal	integers, floats	Represents variable-precision decimal

Cassandra Data Model

CQL Commands – CQL Data Types

double	integers	Represents 64-bit IEEE-754 floating point
float	integers, floats	Represents 32-bit IEEE-754 floating point
inet	strings	Represents an IP address, IPv4 or IPv6
int	integers	Represents 32-bit signed int
text	strings	Represents UTF8 encoded string
timestamp	integers, strings	Represents a timestamp
timeuuid	uuids	Represents type 1 UUID
uuid	uuids	Represents type 1 or type 4

Course Outline

- ❑ **Introduction to NoSQL and Apache Cassandra**
- ❑ **Cassandra Low Level Architecture**
- ❑ **Cassandra Data Model**
- ❑ Compound Primary Keys
- ❑ Advanced Capabilities
- ❑ Composite Partition Keys
- ❑ Indexes and Secondary Indexes
- ❑ Counters
- ❑ Collections
- ❑ Data Consistency and other Cassandra key Features
- ❑ Cassandra Administration Components

Cassandra Data Model

Compound Primary Keys

- For a table with a compound primary key, Cassandra uses a partition key that is either simple or composite. In addition, clustering column(s) are defined.
- Clustering is a storage engine process that sorts data within each partition based on the definition of the clustering columns. Normally, columns are sorted in ascending alphabetical order. Generally, a different grouping of data will benefit reads and writes better than this simplistic choice.
- Remember that data is distributed throughout the Cassandra cluster. An application can experience high latency while retrieving data from a large partition if the entire partition must be read to gather a small amount of data.
- On a physical node, when rows for a partition key are stored in order based on the clustering columns, retrieval of rows is very efficient.
- Grouping data in tables using clustering columns is the equivalent of JOINs in a relational database, but are much more performant because only one table is accessed.

Cassandra Data Model

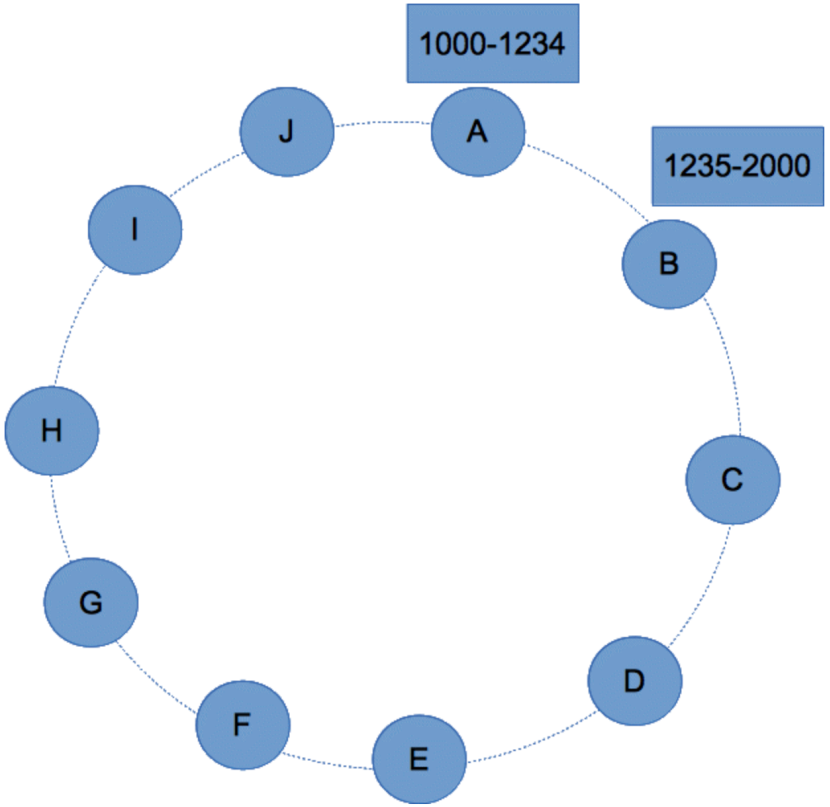
Compound Primary Keys

How does Compound Primary Key Works?

- **C1**: Primary key has only one partition key and no cluster key.
- **(C1, C2)**: Column C1 is a partition key and column C2 is a cluster key.
- **(C1,C2,C3,…)**: Column C1 is a partition key and columns C2, C3, and so on make the cluster key.
- **(C1, (C2, C3,…))**: It is same as 3, i.e., column C1 is a partition key and columns C2,C3,… make the cluster key.
- **((C1, C2,…), (C3,C4,…))**: columns C1, C2 make partition key and columns C3,C4,… make the cluster key.

Cassandra Data Model

Compound Primary Keys – Partition Key



Cassandra Data Model

Compound Primary Keys – Partition Key

- The purpose of a partition key is to identify the partition or node in the cluster that stores that row.
- When data is read or written from the cluster, a function called Partitioner is used to compute the hash value of the partition key.
- This hash value is used to determine the node/partition which contains that row.
- For example, rows whose partition key values range from 1000 to 1234 may reside in node A, and rows with partition key values range from 1235 to 2000 may reside in node B, as shown. If a row contains partition key whose hash value is 1233 then it will be stored in node A.

Cassandra Data Model

Compound Primary Keys – Clustering Key

- The purpose of the clustering key is to store row data in a sorted order.
- The sorting of data is based on columns, which are included in the clustering key.
- This arrangement makes it efficient to retrieve data using the clustering key.

Course Outline

- ❑ **Introduction to NoSQL and Apache Cassandra**
- ❑ **Cassandra Low Level Architecture**
- ❑ **Cassandra Data Model - Intro**
- ❑ **Compound Primary Keys**
- ❑ Advanced Capabilities
- ❑ Composite Partition Keys
- ❑ Indexes and Secondary Indexes
- ❑ Counters
- ❑ Collections
- ❑ Data Consistency and other Cassandra key Features
- ❑ Cassandra Administration Components

Cassandra Data Model

Advanced Capabilities – Time to Live

- Columns and tables support an optional expiration period called TTL (time-to-live)
- Define the TTL value in seconds. Data expires after the data exceeds the TTL period and is then marked with a tombstone
- Expired data continues to be available for read requests during the grace period.
- Normal compaction and repair processes automatically remove the tombstone data.
- TTL precision is one second, which is calculated by the coordinator node. When specifying the TTL, ensure that all nodes in the cluster have synchronized clocks.
- Expiring data uses an additional 8 bytes of memory and disk space to record the TTL and grace period.
- We can set TTL for a specific column or Table overall.

Cassandra Data Model

Advanced Capabilities – Batching

- Batch operations for both single partition and multiple partitions ensure atomicity. An atomic transaction is an indivisible and irreducible series of operations such that either all occur, or nothing occurs. Single partition batch operations are atomic automatically, while multiple partition batch operations require the use of a batchlog to ensure atomicity.

- Batching can be effective for single partition write operations. But batches are often mistakenly used in an attempt to optimize performance.

- Depending on the batch operation, the performance may actually worsen. Some batch operations place a greater burden on the coordinator node and lessen the efficiency of the data insertion.

- The number of partitions involved in a batch operation, and thus the potential for multi-node accessing, can increase the latency dramatically. In all batching, the coordinator node manages all write operations, so that the coordinator node can pose a bottleneck to completion.

Cassandra Data Model

Advanced Capabilities – Indexing

- CREATE INDEX creates a new index on the given table for the named column. Attempting to create an already existing index will return an error unless the IF NOT EXISTS option is used.
- 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.
- Cassandra supports creating an index on most columns, including a clustering column of a compound primary key or on the partition (primary) key itself. Cassandra 2.1 and later supports creating an index on a collection or the key of a collection map.
- Indexing can impact performance greatly. Before creating an index, be aware of when and when not to create the index.

Cassandra Data Model

Advanced Capabilities – Secondary Indexing

- Using CQL, you can create an index on a column after defining a table.
- Secondary indexes are used to query a table using a column that is not normally queryable.
- Secondary indexes are tricky to use and can impact performance greatly. The index table is stored on each node in a cluster, so a query involving a secondary index can rapidly become a performance nightmare if multiple nodes are accessed. A general rule of thumb is to index a column with low cardinality of few values.

Create a table in your CQL with the below structure

cqlsh> **CREATE TABLE** navaneeth_ks.rank_by_year_and_name (race_year int, race_name text, cyclist_name text, rank int, **PRIMARY KEY** ((race_year, race_name), rank));

Insert the values in below format

race_year	race_name	rank	cyclist_name
2015	Tour of Japan - Stage 4 - Minami > Shinshu	1	Benjamin PRADES
2015	Tour of Japan - Stage 4 - Minami > Shinshu	2	Adam PHELAN
2015	Tour of Japan - Stage 4 - Minami > Shinshu	3	Thomas LEBAS

Cassandra Data Model

Advanced Capabilities – Secondary Indexing

- Both race_year and race_name must be specified as these columns comprise the partition key.

  cqlsh> SELECT * FROM cycling.rank_by_year_and_name WHERE race_year=2015 AND race_name='Tour of Japan - Stage 4 - Minami > Shinshu';

- A logical query to try is a listing of the rankings for a particular year. Because the table has a composite partition key, this query will fail if only the first column is used in the conditional operator.

  cqlsh> SELECT * FROM cycling.rank_by_year_and_name WHERE race_year=2015;

```
cqlsh:cycling> SELECT * from rank_by_year_and_name where race_year=2015;
InvalidRequest: code=2200 [Invalid query] message="Partition key parts: race_name must be restricted as other parts are"
```

- An index is created for the race year, and the query will succeed. An index name is optional and must be unique within a keyspace. If you do not provide a name, Cassandra will assign a name like race_year_idx.

  cqlsh> CREATE INDEX ryear ON cycling.rank_by_year_and_name (race_year);
  SELECT * FROM cycling.rank_by_year_and_name WHERE race_year=2015;

Cassandra Data Model

Advanced Capabilities – Secondary Indexing

- A clustering column can also be used to create an index. An index is created on rank, and used in a query

```
cqlsh> CREATE INDEX rrank ON cycling.rank_by_year_and_name (rank);
SELECT * FROM cycling.rank_by_year_and_name WHERE rank = 1;
```

Cassandra Data Model

Advanced Capabilities – Allow Filtering

- If a query does not specify the values for all the columns from the primary key in the ‘where’ clause, Cassandra will not execute it
- You can use execute queries that use a secondary index without ALLOW FILTERING – more on that later.
- The ‘ALLOW FILTERING’ clause in Cassandra CQL provides greatly increased flexibility of querying. However, this flexibility comes at a substantial performance cost that should be aware of before using ‘ALLOW FILTERING’.

Cassandra Data Model

Advanced Capabilities – Materialized View

- Materialized views handle automated server-side denormalization, removing the need for client side handling of this denormalization and ensuring eventual consistency between the base and view data.
- This denormalization allows for very fast lookups of data in each view using the normal Cassandra read path.
- Materialized views maintain a correspondence of one CQL row each in the base and the view, so we need to ensure that each CQL row which is required for the views will be reflected in the base table's primary keys.
- When a deletion occurs, the materialized view will query all of the deleted values in the base table and generate tombstones for each of the materialized view rows, because the values that need to be tombstoned in the view are not included in the base table's tombstone.
- When a materialized view is created against a table which has data already, a building process will be kicked off to populate the materialized view. As such, materialized views can be created on existing tables, but there will be a period during which queries against the materialized view may not return all results

Cassandra Data Model

Advanced Capabilities – Materialized View

- When a base view is altered, the materialized view is updated as well. If the materialized view has a SELECT * statement, any added columns will be included in the materialized view's columns. Any deleted columns which are part of the SELECT statement will be removed from the materialized view. If a column in the base table is altered, the same alteration will occur in the view table. If the base table is dropped, any associated views will also be dropped.

- Materialized views do not have the same write performance characteristics that normal table writes have.

- If there will be a large number of partition tombstones, the performance may suffer; the materialized view must query for all of the current values and generate a tombstone for each of them. The materialized view will have one tombstone per CQL row deleted in the base table

- Currently, only simple SELECT statements are supported

Course Outline

- **Introduction to NoSQL and Apache Cassandra**
- **Cassandra Low Level Architecture**
- **Cassandra Data Model**
- **Compound Primary Keys**
- Advanced Capabilities
- Indexes and Secondary Indexes
- Counters
- Collections
- Data Consistency and other Cassandra key Features
- Cassandra Administration Components

Advanced Capabilities

Indexes

- Cassandra doesn’t have a Single Point of Failures(SPoF), supports multi-region, good for read and write ops, flexible about read and write consistency levels and scales linearly.

- We should use Cassandra based on your data access patterns, **so if you need a flexible database for ad-hoc queries or adaptable enough for constantly database model changes, you should consider other options.**

- Cassandra is a column-oriented DB and it’s really powerful when you have your data queries already defined.

Advanced Capabilities

Indexes

- Cassandra is a primary key database, which means your data is persisted and organized around a cluster based on the hash value (the partition key) of the primary key. For tables that have more than one PK, Cassandra considers only the first part of the PK as it's partition key.

- Cassandra has also the concept of secondary indexes. In relational databases, you could have many indexes in a given table, the cost of have a secondary index is associated with write operations, not for read operations. In Cassandra this is not true.

- Secondary indexes in Cassandra could be useful and tempting when your data model changed and you need to query based on a new column.

Advanced Capabilities

Indexes

- Imagine your e-commerce website got more request than the usual due to some special days. WE may need to suffer latency issues due to more number of customers access our portals for offers or deals.

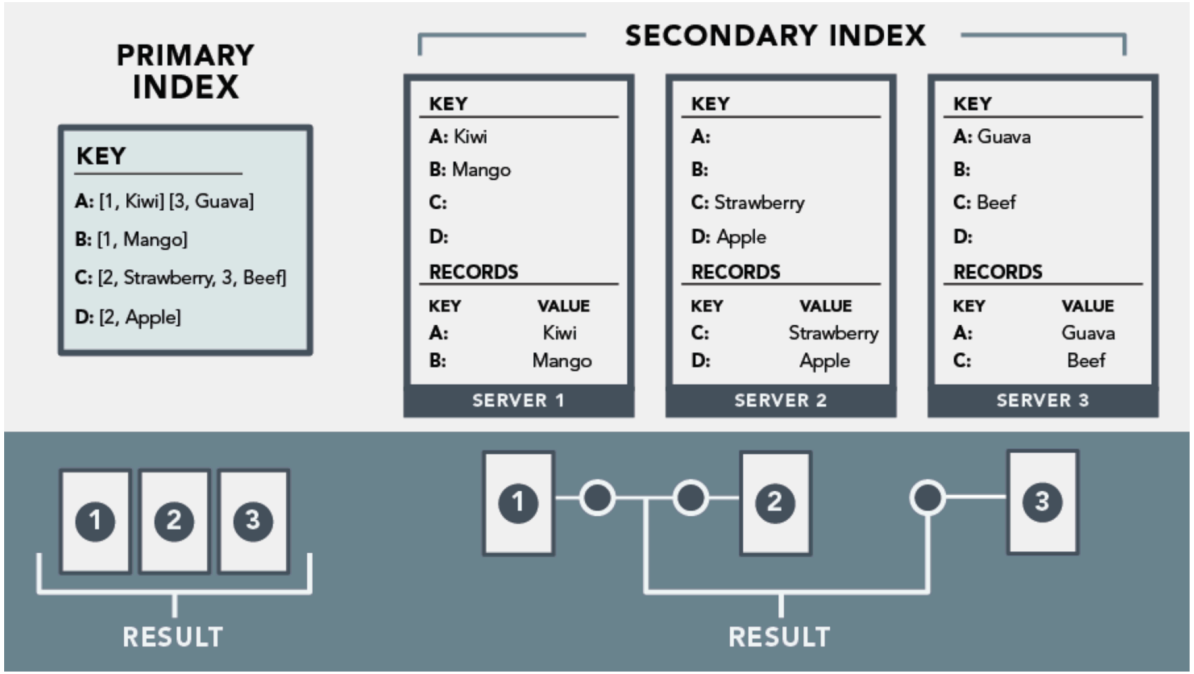
- To have more queries running, we can reduce the read latency as a first choice. This will reduce the load to some scale. Another options are indexing, especially managing secondary indexing.

- Secondary indexes are indexes built over column values. In other words, let's say you have a user table, which contains a user's email. The primary index would be the user ID, so if you wanted to access a particular user's email, you could look them up by their ID. However, to solve the inverse query, given an email, fetch the user ID. Here requires a secondary index.

Advanced Capabilities

Indexes

- The primary Index is global for Cassandra whereas the Secondary Index is local. Primary Index is created on primary key.



Advanced Capabilities

Indexes

- Consider we re running Cassandra on a ring of five machines, with a primary index of user IDs and a secondary index of user emails. If you were to query for a user by their ID, or by their primary indexed key—any machine in the ring would know which machine has a record of that user.

- One query, one read from disk. However to query a user by their email, or their secondary indexed value, each machine has to query its own record of users. One query, five reads from disk.

- By either scaling the number of users system wide, or by scaling the number of machines in the ring, the noise to signal-to-ratio increases and the overall efficiency of reading drops - in some cases to the point of timing out on API calls

Advanced Capabilities

Allow Filtering

- Select queries in Cassandra look a lot like select queries from a relational database. However, they are significantly more restricted. The attributes allowed in ‘where’ clause of Cassandra query must include the full partition key and additional clauses may only refer the clustering key columns or a secondary index of the table being queried.

- Requiring the partition key attributes in the ‘where’ helps Cassandra to maintain constant result-set retrieval time as the cluster is scaled-out by allowing Cassandra to determine the partition, and thus the node (and even data files on disk), that the query must be directed to.

Advanced Capabilities

Allow Filtering

- If a query does not specify the values for all the columns from the primary key in the ‘where’ clause, Cassandra will not execute it and give the following warning :

  ‘*InvalidRequest: Error from server: code=2200 [Invalid query] message=”Cannot execute this query as it might involve data filtering and thus may have unpredictable performance. If you want to execute this query despite the performance unpredictability, use ALLOW FILTERING*”

- You can use execute queries that use a secondary index without ALLOW FILTERING

- The reason behind this warning is that when the complete partition key is not included in the WHERE clause, there is no way for Cassandra to identify the node which contains the required results, and thus it will need to scan the complete dataset on each node to ensure it has found the required data.

Advanced Capabilities

Allow Filtering

- Consider a Scenario

```
CREATE TABLE users (
username text PRIMARY KEY,
firstname text,
lastname text,
birth_year int,
country text
)
```

- Create an Index on birth Year Column

```
CREATE INDEX ON users(birth_year);
```

Then the following queries are valid:
Query 1: SELECT * FROM users;
Query 2: SELECT * FROM users WHERE birth_year = 1981;

Advanced Capabilities

Allow Filtering

- The Following Query will be rejected

  SELECT * FROM users WHERE birth_year = 1981 AND country = ‘AU’;

This is because Cassandra cannot guarantee that it won’t have to scan a large amount of data even if the result of the query is small. Typically, it will scan all the index entries for users born in 1981 even if only a handful are actually from Australia. However, if you “know what you are doing”, you can force the execution of this query by using ALLOW FILTERING and so the following query is valid:

- The above query with ALLOW FILTERING will work

  SELECT * FROM users WHERE birth_year = 1981 AND country = ‘AU’ ALLOW FILTERING;

Course Outline

- **Introduction to NoSQL and Apache Cassandra**
- **Cassandra Low Level Architecture**
- **Cassandra Data Model**
- **Compound Primary Keys**
- **Advanced Capabilities**
- **Indexes and Secondary Indexes**
- Counters
- Collections
- Data Consistency and other Cassandra key Features
- Cassandra Administration Components

Compiled by Navaneetha Babu C

Cassandra Data Modelling – Counters

Counters

- A counter is a special column used to store an integer that is changed in increments. Counters are useful for many data models.

Some Examples are

- To keep track of the number of web page views received on a company website
- To keep track of the number of games played online or the number of players who have joined an online game

- Lets create a table with counter

```
CREATE TABLE WebLogs (
page_id uuid,
page_name Text,
insertion_time timestamp,
page_count counter,
PRIMARY KEY (page_id, insertion_time)
);
```

Cassandra Data Modelling – Counters

Counters

- The previous Create query will fail with the below error

  ERROR : – InvalidRequest: Error from server: code=2200 [Invalid query] message=”Cannot mix counter and non counter columns in the same table”

- We have to use composite keys in the create to place the counter columns. This is mandatory

```
CREATE TABLE WebLogs (
page_id uuid,
page_name Text,
insertion_time timestamp,
page_count counter,
PRIMARY KEY ((page_id,page_name), insertion_time
);
```

- Lets Insert some Data

  insert into weblogs (page_id , page_name , insertion_time , page_count) values(uuid(),’test.com’,dateof(now()),0) ;

Cassandra Data Modelling – Counters

Counters

- There will be an error in the insert statement, because we are trying to insert values to counter column, which are not allowed. It is required to use Update statement

  update weblogs set page_count = page_count + 1 where page_id =uuid() and page_name =’test.com’ and insertion_time =dateof(now());

  select * from weblogs;

  page_id | page_name | insertion_time | page_count
  ——————————————+———+——————8372cee6-1d04-41f7-a70d-98fdd9036448 | test.com | 2017-01-05 05:19:31+0000 | 1

- Lets Insert More records and see the counters gets automatically increasing

  update weblogs set page_count = page_count + 1 where page_id =8372cee6-1d04-41f7-a70d-98fdd9036448 and page_name =’test.com’ and insertion_time =’2017-01-05 05:19:31+0000′;

  se**lect * from weblogs;**

Cassandra Data Modelling – Counters

Counters

More Info about Counters

- A counter column cannot be indexed or deleted..

- To load data into a counter column, or to increase or decrease the value of the counter, use the UPDATE command. Cassandra rejects USING TIMESTAMP or USING TTL when updating a counter column.

- To create a table having one or more counter columns, use this:
  Use CREATE TABLE to define the counter and non-counter columns. Use all non-counter columns as part of the PRIMARY KEY definition.

- You can create table with multiple counter columns.

Cassandra Data Modelling – Counters

Counters

- Counters will reduce when we delete the data.

```
delete from weblogs where  page_id =8372cee6-1d04-41f7-a70d-98fdd9036448 and
page_name =’test.com’ and insertion_time =’2017-01-05 05:19:31+0000′;

select * from weblogs;
```

Course Outline

- ❑ **Introduction to NoSQL and Apache Cassandra**
- ❑ **Cassandra Low Level Architecture**
- ❑ **Cassandra Data Model**
- ❑ **Compound Primary Keys**
- ❑ **Advanced Capabilities**
- ❑ **Indexes and Secondary Indexes**
- ❑ **Counters**
- ❑ Collections
- ❑ Data Consistency and other Cassandra key Features
- ❑ Cassandra Administration Components

Cassandra Data Modelling – Collections

Collections

Collections give you three types:

- Set
- List
- Map

Cassandra provides collection types as a way to group and store data together in a column

Each allow for dynamic updates

Collections requires serializations

A collection is appropriate if the data for collection storage is limited. If the data has unbounded growth potential, like messages sent or sensor events registered every second, do not use collections. Instead, use a table with a compound primary key where data is stored in the clustering columns.

Cassandra Data Modelling – Collections

Limitations of Collections

- Never insert more than 2 billion items in a collection, as only that number can be queried.
- The maximum number of keys for a map collection is 65,535.
- The maximum size of an item in a list or a map collection is 2GB.
- The maximum size of an item in a set collection is 65,535 bytes.
- Keep collections small to prevent delays during querying. Collections cannot be "sliced"; Cassandra reads a collection in its entirety, impacting performance. Thus, collections should be much smaller than the maximum limits listed. The collection is not paged internally.
- Lists can incur a read-before-write operation for some insertions. Sets are preferred over lists whenever possible.

Cassandra Data Modelling – Collections

Examples

- Create the users table with one column as set, another as list and one more as MAP

```
CREATE TABLE users(username TEXT PRIMARY KEY, emails SET<TEXT>, phones
MAP<TEXT,TEXT>, top_cities LIST<TEXT>);
```

- Insert the values to collections

```
INSERT INTO users(username, emails, phones, top_cities) VALUES (‘navaneeth',
{‘navaneeth@yahoo.com', ‘navaneeth@gmail.com.com'}, {'home' : '999-9999', 'mobile' : '000-0000'}, ['New
York', 'Paris’]);
```

- In the collections, empty collections are same as null

```
INSERT INTO users(username, emails, phones, top_cities) VALUES (‘ananth', { }, { }, [ ]);

SELECT * from users;
```

Cassandra Data Modelling – Collections

Examples : Update Collection Columns

```
UPDATE users SET emails = {‘ananth@gmail.com'} WHERE username = ‘ananth’;

UPDATE users SET phones = {'home' : '123-45678'} WHERE username = ‘ananth’;

UPDATE users SET top_cities = ['London', 'Tokyo'] WHERE username = ‘ananth;

SELECT * from users;
```

Cassandra Data Modelling – Collections

Examples : Collection Expressions

Collection elements can be added with + and removed with -

```
UPDATE users SET emails = emails + {‘navaneeth@hotmail.com'} WHERE username = ‘navaneeth’;

UPDATE users SET emails = emails - {‘navaneeth@yahoo.com'} WHERE username = ‘navaneeth’;

UPDATE users SET phones = phones + {'office' : '333-3333'} WHERE username = ‘navaneeth’;

SELECT * from users;
```

Cassandra Data Modelling – Collections

Examples : Collection Expressions

To remove map elements only the relevant keys need to be given (as a set).

```
UPDATE users SET phones = phones - {'home'} WHERE username = ‘navaneeth’;

SELECT * from users;
```

List elements can be either prepended or appended.

```
UPDATE users SET top_cities = top_cities + [‘Chennai'] WHERE username = ‘navaneeth’;

UPDATE users SET top_cities = ['Sunnyvale'] + top_cities WHERE username = ‘navaneeth’;

UPDATE users SET top_cities = top_cities - ['Paris', 'New York'] WHERE username =
‘navaneeth’;

SELECT * from users;
```

Cassandra Data Modelling – Collections

Examples : Collection Expressions

Update Map and List Elements

```
UPDATE users SET phones['mobile'] = '111-1111' WHERE username = ‘navaneeth’;

UPDATE users SET phones['mobile'] = '345-6789' WHERE username = ‘analth' IF phones['mobile'] = null;

SELECT * from users;
```

Lists allow referencing elements by index

```
UPDATE users SET top_cities[0] = 'San Francisco' WHERE username = ‘ananth’;

UPDATE users SET top_cities[1] = 'Mumbai' WHERE username = ‘ananth' IF top_cities[1] = 'Tokyo’;

SELECT * from users;
```

Cassandra Data Modelling – Collections

Examples : Indexes in Collections

- For set and list collections, create an index on the column name.

```
CREATE INDEX top_city_inx on users(top_cities)
```

- For map collections, create an index on the map key, map value, or map entry.

```
CREATE INDEX phones_inx on users( KEYS(phones) );

CREATE INDEX phones_inx_all on users( ENTRIES(phones) );
```

Course Outline

- ❑ **Course Introduction**
- ❑ **Introduction to NoSQL and Apache Cassandra**
- ❑ **Cassandra Low Level Architecture**
- ❑ **Cassandra Data Model**
- ❑ **Compound Primary Keys**
- ❑ **Advanced Capabilities**
- ❑ **Indexes and Secondary Indexes**
- ❑ **Counters**
- ❑ **Collections**
- ❑ Data Consistency and other Cassandra key Features
- ❑ Cassandra Administration Components

Cassandra Data Consistency

Write Consistency

- Consistency levels in Cassandra can be configured to manage availability versus data accuracy.

- Configure consistency for a session or per individual read or write operation.

ALL
- A write must be written to the commit log and memtable on all replica nodes in the cluster for that partition.
- Provides the highest consistency and the lowest availability of any other level.

EACH_QUORUM
- Strong consistency. A write must be written to the commit log and memtable on a quorum of replica nodes in each datacenter.
- Used in multiple datacenter clusters to strictly maintain consistency at the same level in each datacenter. For example, choose this level if you want a write to fail when a datacenter is down and the QUORUM cannot be reached on that datacenter.

Cassandra Data Consistency

Write Consistency

QUORUM

- A write must be written to the commit log and memtable on a quorum of replica nodes across all datacenters.
- Used in either single or multiple datacenter clusters to maintain strong consistency across the cluster. Use if you can tolerate some level of failure.

LOCAL_QUORUM

- Strong consistency. A write must be written to the commit log and memtable on a quorum of replica nodes in the same datacenter as the coordinator. Avoids latency of inter-datacenter communication.
- Used in multiple datacenter clusters with a rack-aware replica placement strategy, such as NetworkTopologyStrategy, and a properly configured snitch. Use to maintain consistency locally (within the single datacenter). Can be used with SimpleStrategy.

Cassandra Data Consistency

Write Consistency

ONE
- A write must be written to the commit log and memtable of at least one replica node.
- Satisfies the needs of most users because consistency requirements are not stringent.

TWO
- A write must be written to the commit log and memtable of at least two replica node.

THREE
- A write must be written to the commit log and memtable of at least three replica node.

LOCAL_ONE
- A write must be sent to, and successfully acknowledged by, at least one replica node in the local datacenter.
- In a multiple datacenter clusters, a consistency level of ONE is often desirable, but cross-DC traffic is not. LOCAL_ONE accomplishes this. For security and quality reasons, you can use this consistency level in an offline datacenter to prevent automatic connection to online nodes in other datacenters if an offline node goes down.

Cassandra Data Consistency

Write Consistency

ANY

- A write must be written to at least one node. If all replica nodes for the given partition key are down, the write can still succeed after a hinted handoff has been written. If all replica nodes are down at write time, an ANY write is not readable until the replica nodes for that partition have recovered.
- Provides low latency and a guarantee that a write never fails. Delivers the lowest consistency and highest availability.

Cassandra Data Consistency

Read Consistency

ALL

- Returns the record after all replicas have responded. The read operation will fail if a replica does not respond.
- Provides the highest consistency of all levels and the lowest availability of all levels.

QUOROM

- Returns the record after a quorum of replicas from all datacenters has responded.
- Used in either single or multiple datacenter clusters to maintain strong consistency across the cluster. Ensures strong consistency if you can tolerate some level of failure.

LOCAL_QUOROM

- Returns the record after a quorum of replicas in the current datacenter as the coordinator has reported. Avoids latency of inter-datacenter communication.
- Used in multiple datacenter clusters with a rack-aware replica placement strategy (NetworkTopologyStrategy) and a properly configured snitch. Fails when using SimpleStrategy.

Cassandra Data Consistency

Read Consistency

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.
- Provides the highest availability of all the levels if you can tolerate a comparatively high probability of stale data being read. The replicas contacted for reads may not always have the most recent write.

TWO
- Returns the most recent data from two of the closest replicas.

THREE
- Returns the most recent data from three of the closest replicas

LOCAL_ONE
- Returns a response from the closest replica in the local datacenter.
- Same usage as described in the table about write consistency levels.

Cassandra Data Consistency

Read Consistency

SERIAL

- Allows reading the current (and possibly uncommitted) state of data without proposing a new addition or update. If a SERIAL read finds an uncommitted transaction in progress, it will commit the transaction as part of the read. Similar to QUORUM.
- To read the latest value of a column after a user has invoked a LWT to write to the column, use SERIAL. Cassandra then checks the inflight lightweight transaction for updates and, if found, returns the latest data

LOCAL_SERIAL

- Same as SERIAL, but confined to the datacenter. Similar to LOCAL_QUORUM.

Quorom Calculation :
quorum = (sum_of_replication_factors / 2) + 1

Course Outline

- ❑ **Course Introduction**
- ❑ **Introduction to NoSQL and Apache Cassandra**
- ❑ **Cassandra Low Level Architecture**
- ❑ **Cassandra Data Model - Intro**
- ❑ **Compound Primary Keys**
- ❑ **Advanced Capabilities**
- ❑ **Indexes and Secondary Indexes**
- ❑ **Counters**
- ❑ **Collections**
- ❑ **Data Consistency and other Cassandra key Features**
- ❑ Cassandra Administration Components

Data Consistency

Data Caching In Cassandra

- Key caching is enabled by default in Cassandra, and high levels of key caching are recommended for most scenarios.

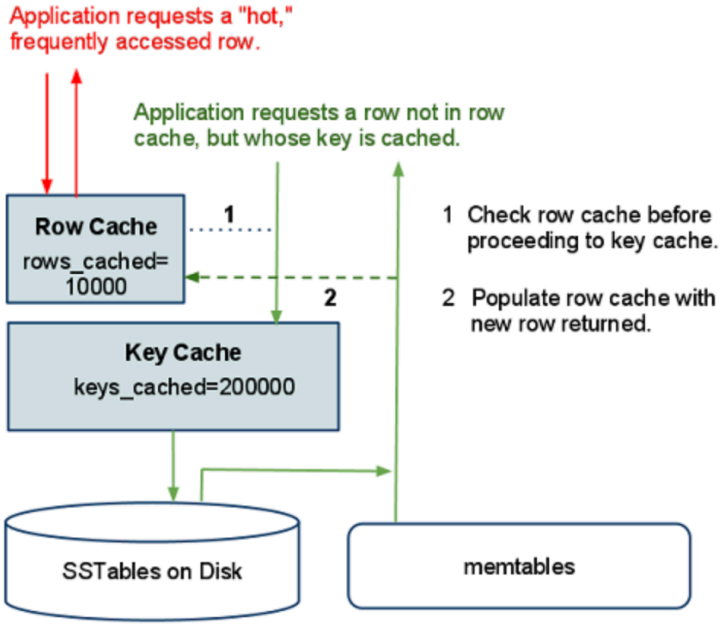
- Cases for row caching are more specialized, but whenever it can coexist peacefully with other demands on memory resources, row caching provides the most dramatic gains in efficiency.

- The key cache holds the location of keys in memory on a per-column family basis. For column family level read optimizations, turning this value up can have an immediate impact as soon as the cache warms. Key caching is enabled by default, at a level of 200,000 keys.

Data Consistency

Data Caching In Cassandra

- Unlike the key cache, the row cache holds the entire contents of the row in memory. It is best used when you have a small subset of data to keep hot and you frequently need most or all of the columns returned. For these use cases, row cache can have substantial performance benefits.



Data Consistency

Data Caching In Cassandra

- One read operation hits the row cache, returning the requested row without a disk seek. The other read operation requests a row that is not present in the row cache but is present in the key cache. After accessing the row in the SSTable, the system returns the data and populates the row cache with this read operation.

CREATE TABLE users (userid text PRIMARY KEY, first_name text, last_name text,) WITH **caching = { 'keys' : 'NONE', 'rows_per_partition' : '120' };**

Data Consistency

Compaction

- The compaction process merges keys, combines columns, evicts tombstones, consolidates SSTables, and creates a new index in the merged SSTable.

- In the cassandra.yaml file, you configure these global compaction parameters:

  snapshot_before_compaction concurrent_compactors

  compaction_throughput_mb_per_sec

- The different Compaction Strategies are

  LeveledCompactionStrategy

  SizeTieredCompactionStrategy

  TimeWindowCompactionStrategy

  DataTieredCompactionStrategy

Data Consistency

Compaction

- Periodic compaction is essential to a healthy Cassandra database because Cassandra does not insert/update in place. As inserts/updates occur, instead of overwriting the rows, Cassandra writes a new timestamped version of the inserted or updated data in another SSTable.

- Cassandra also does not delete in place because SSTables are immutable. Instead, Cassandra marks data to be deleted using a tombstone. Tombstones exist for a configured time period defined by the gc_grace_seconds value set on the table.

- Over time, many versions of a row might exist in different SSTables. Each version has a different set of columns stored. As SSTables accumulate, more and more SSTables must be read in order to retrieve an entire row of data.

Data Consistency

Compaction

- Compaction merges the data in each SSTable by partition key, selecting the latest data for storage based on its timestamp. Because rows are sorted by partition key within each SSTable, the merge process does not use random I/O and is performant.

- After evicting tombstones and removing deleted data, columns, and rows, the compaction process consolidates SSTables into a new single SSTable file. The old SSTable files are deleted as soon as any pending reads finish using the files.

- During compaction, there is a temporary spike in disk space usage and disk I/O because the old and new SSTables co-exist. Disk space occupied by old SSTables becomes available for reuse when the new SSTable is ready.

Compaction

LeveledCompactionStrategy

- The leveled compaction strategy creates SSTables of a fixed, relatively small size (160 MB by default) that are grouped into levels.
- Within each level, SSTables are guaranteed to be non-overlapping. Each level (L0, L1, L2 and so on) is 10 times as large as the previous. Disk I/O is more uniform and predictable on higher than on lower levels as SSTables are continuously being compacted into progressively larger levels.
- At each level, row keys are merged into non-overlapping SSTables in the next level. This process can improve performance for reads, because Cassandra can determine which SSTables in each level to check for the existence of row key data.

- This is good for read intensive applications

Data Consistency

Compaction

SizeTieredCompaction

- The SizeTieredCompactionStrategy (STCS) initiates compaction when a set number (default is 4) of similar-sized SSTables have accumulated. Compaction merges the SSTables to create one larger SSTable. This is for Write Intensive Applications

- At any given time, several SSTables of varying sizes are present. While this strategy works quite well to compact a write-intensive workload, when reads are needed, several SSTables still must be retrieved to find all the data for a row. There is no guarantee that a row's data will be restricted to a small number of SSTables.
- As the largest SSTables grow in size, the amount of memory needed for compaction to hold both the new and old SSTables simultaneously can outstrip a typical amount of RAM on a node.

Data Consistency

Compaction

TimeWindowLevelCompaction

- It is suitable for timeseries based designs. The Time Window Compaction Strategy is designed to work on time series data. It compactes SSTables within a configured time window. TWCS utilizes STCS to perform these compactions. At the end of each time window, all SSTables are compacted to a single SSTable so there is one SSTable for a time window. TWCS also effectively purges data when configured with time to live by dropping complete SSTables after TTL expiry.

- SSTables are merged when a certain minimum threshold of number of SSTables is reached within a configurable time interval.

Data Consistency

Compaction

Examples

ALTER TABLE users WITH compaction = { 'class' : 'LeveledCompactionStrategy’ }

ALTER TABLE users WITH compaction = {'class' : 'SizeTieredCompactionStrategy', 'min_threshold' : 6 }

ALTER TABLE users WITH compaction = {'compaction_window_size': '1', 'compaction_window_unit': 'MINUTES', 'class': 'org.apache.cassandra.db.compaction.TimeWindowCompactionStrategy'}

Course Outline

- ❑ **Course Introduction**
- ❑ **Introduction to NoSQL and Apache Cassandra**
- ❑ **Cassandra Low Level Architecture**
- ❑ **Cassandra Data Model - Intro**
- ❑ **Compound Primary Keys**
- ❑ **Advanced Capabilities**
- ❑ **Indexes and Secondary Indexes**
- ❑ **Counters**
- ❑ **Collections**
- ❑ **Data Consistency and other Cassandra key Features**
- ❑ Cassandra Administration Components

Cassandra Adminstration components

Hardware and Capacity planning

- Like most databases, Cassandra throughput improves with more CPU cores, more RAM, and faster disks. While Cassandra can be made to run on small servers for testing or development environments, a minimal production server requires at least 2 cores, and at least 8GB of RAM.

- Typical production servers have 8 or more cores and at least 32GB of RAM upto 256GB RAM

- While running Cassandra, we need to think on both RAM and Disk access, because Memtables are stored in RAM and SSTables are stored in DIsks

Cassandra Adminstration components

Hardware and Capacity planning

CPU Cores

- Cassandra is highly concurrent, handling many simultaneous requests (both read and write) using multiple threads running on as many CPU cores as possible.

- The Cassandra write path tends to be heavily optimized (writing to the commitlog and then inserting the data into the memtable), so writes, in particular, tend to be CPU bound.

- Consequently, adding additional CPU cores often increases throughput of both reads and writes.

Cassandra Adminstration components

Hardware and Capacity planning

RAM Memory

- Cassandra runs within a Java VM, which will pre-allocate a fixed size heap (java's Xmx system parameter). In addition to the heap, Cassandra will use significant amounts of RAM offheap for compression metadata, bloom filters, row, key, and counter caches, and an in process page cache.

- Cassandra will take advantage of the operating system's page cache, storing recently accessed portions files in RAM for rapid re-use.

- Cassandra heap should not be more than 50% of your system RAM

Cassandra Adminstration components

Hardware and Capacity planning

Disk

- Cassandra persists data to disk for two very different purposes. The first is to the commitlog when a new write is made so that it can be replayed after a crash or system shutdown. The second is to the data directory when thresholds are exceeded and memtables are flushed to disk as SSTables.

- Commitlogs receive every write made to a Cassandra node and have the potential to block client operations, but they are only ever read on node start-up.

- SSTable (data file) writes on the other hand occur asynchronously, but are read to satisfy client look-ups. SSTables are also periodically merged and rewritten in a process called compaction. The data held in the commitlog directory is data that has not been permanently saved to the SSTable data directories - it will be periodically purged once it is flushed to the SSTable data files.

Cassandra Adminstration components

Hardware and Capacity planning

- Cassandra performs very well on both spinning hard drives and solid state disks. In both cases, Cassandra's sorted immutable SSTables allow for linear reads, few seeks, and few overwrites, maximizing throughput for HDDs and lifespan of SSDs by avoiding write amplification.

- In most cases, Cassandra is designed to provide redundancy via multiple independent, inexpensive servers.

- Using NFS or a SAN for data directories is an antipattern and should typically be avoided. Use Dedicated Drives

Cassandra Adminstration components

Hardware and Capacity planning

- RAID is preferred at one Disk which is used for OS and logs
- JBOD is preferred for storing the SSTables
- RAID is preferred for Storing CommitLogs
- Many large users of Cassandra run in various clouds, including AWS, Azure, and GCP

Cassandra Adminstration components

Hardware and Capacity planning

Enable Cgroups in cluster

- **The following resource limits can be configured after enabling cgroups** – **Memory Hard Limit**
  – If a process exceeds this limit, the kernel swaps out some of the process's memory; if it cannot do so, the process will be killed

- **Memory Soft Limit**
  – When memory contention exists on the host, the OS targets the process to not exceed this limit

- **CPU Shares**
  When CPU contention exists on the host, processes with higher CPU shares will be given more CPU time

- **I/O Weight**
  Specify the proportion of I/O access available to the read requests performed by a process

Cassandra Adminstration components

Hardware and Capacity planning

Typical configurations for Cassandra nodes
Midline: deep storage, 1Gb Ethernet
– 16 x 3TB SATA II hard drives, in a non-RAID, JBOD* configuration – 1 or 2 of the 16 drives for the OS, with RAID-1 mirroring
– 2 x 8-core 3.0GHz CPUs, 15MB cache
– 256GB RAM
– 2x1 Gigabit Ethernet

High-end: high memory, spindle dense, 10Gb Ethernet
– 24 x 1TB Nearline/MDL SAS hard drives, in a non-RAID, JBOD* configuration
– 2 x 8-core 3.0GHz CPUs, 15MB cache
– 512GB RAM (or more)
– 1x10 Gigabit Ethernet

Cassandra Adminstration components

Hardware and Capacity planning

- **Hex- and octo-core CPUs are commonly available**
- **Hyper-threading and quick-path interconnect (QPI) should be enabled**
- **Cassandra nodes are typically disk- and network-I/O bound**
  – Therefore, top-of-the-range CPUs are usually not necessary
- **In general, more spindles (disks) is better**
- **Use 3.5 inch disks**
  – Faster, cheaper, higher capacity than 2.5" disks
- **7,200 RPM SATA/SATA II drives are fine** – No need to buy 15,000 RPM drives
- **8 x 1.5TB drives is likely to be better than 6 x 2TB drives**
  – Different tasks are more likely to be accessing different disks

Cassandra Adminstration components

Hardware and Capacity planning

Blade servers are not recommended

– Failure of a blade chassis results in many nodes being unavailable
– Individual blades usually have very limited RAM and hard disk capacity
– Network interconnection between the chassis and top-of-rack switch can become a bottleneck

Cassandra Adminstration components

NodeTool

- Cassandra nodetool is an utility to Manage the Cassandra Cluster.
- NodeTool Components and commands are

Assasinate - Forcefully removes a dead node without re-replicating any data. It is a last resort tool if you cannot successfully use nodetool removenode command

Usage – **nodetool assassinate <ip_address>**

```
cassandra@mongodbPoc2:~/node1/bin$ ./nodetool assassinate 104.42.177.52
cassandra@mongodbPoc2:~/node1/bin$ ./nodetool status
Datacenter: Cassandra
=====================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving/Stopped
--  Address        Load       Tokens       Owns (effective)  Host ID                               Rack
UN  23.99.66.193   2.07 GiB   8            100.0%            0403538f-62e0-4843-9798-70e04b65c5e7  rack1
```

Cassandra Adminstration components

NodeTool

Bootstrap - Monitor/manage node's bootstrap process

Usage – **nodetool bootstrap resume**

```
[root@cassandra-3 ~]# nodetool bootstrap resume
Node is already bootstrapped.
```

Cleanup - Cleans up keyspaces and partition keys no longer belonging to a node.

- Use this command to remove unwanted data after adding a new node to the cluster. Cassandra does not automatically remove data from nodes that lose part of their partition range to a newly added node. Run nodetool cleanup on the source node and on neighboring nodes that shared the same subrange after the new node is up and running.

Usage **nodetool cleanup navaneeth_ks**

Cassandra Adminstration components

NodeTool

Cleanup - Cleans up keyspaces and partition keys no longer belonging to a node.

- Use this command to remove unwanted data after adding a new node to the cluster. It frees partition keys no longer belonging to a node, Cassandra does not automatically remove data from nodes that lose part of their partition range to a newly added node. Run nodetool cleanup on the source node and on neighboring nodes that shared the same subrange after the new node is up and running.
- Failure to run this command after adding a node causes Cassandra to include the old data to rebalance the load on that node. Running the nodetool cleanup command causes a temporary increase in disk space usage proportional to the size of your largest SSTable. Disk I/O occurs when running this command.

Usage - **nodetool cleanup navaneeth_ks**

Cassandra Adminstration components

NodeTool

Compact – Forces compaction on entire keyspace or tables

Usage – **nodetool compact –st <startingToken> -et <endingToken> keyspace**

Clearsnapshot – Removes one or more saved snapshots

Usage – **nodetool clearsnapshot –t <snapshot_name> keyspace**

Create a snapshot in Cassandra using **nodetool snapshot keyspace** command

```
[root@cassandra-1 ~]# nodetool snapshot navaneeth_ks
Requested creating snapshot(s) for [navaneeth_ks] with snapshot name [1602113437588]
Snapshot directory: 1602113437588
[root@cassandra-1 ~]#
```

```
Snapshot directory: 1602113437588
[[root@cassandra-1 ~]# nodetool clearsnapshot 1602113437588 navaneeth_ks
Requested clearing snapshot(s) for [1602113437588, navaneeth_ks]
[root@cassandra-1 ~]#
```

Cassandra Adminstration components

NodeTool

Compactionhistory - Provides the history of compaction operations

Usage – **nodetool compactionhistory**

Compactionstats - Provide statistics about a compaction.

Usage – **nodetool compactionstats –H**

Decommission - Deactivates a node by streaming its data to another node.

Usage - **nodetool decommission –h <ipaddress>**

DescribeCluster - Provide the name, snitch, partitioner and schema version of a cluster

Usage – **nodetool describecluster**

Cassandra Adminstration components

NodeTool

Describering - Provides the partition ranges of a keyspace.

Usage – **nodetool describering navaneeth_ks**

Disableautocompaction - Disables autocompaction for a keyspace and one or more tables

Usage - **nodetool disableautocompaction navaneeth_ks**

Disablebackup - Disables incremental backup. This should happen before shutdown Cassandra.

Usage – **nodetool disablebackup**

DisableGossip - Disables the gossip protocol. This should happen before shutdown Cassandra.

Usage – **nodetool disablegossip**

Cassandra Adminstration components

NodeTool

DisableHandoff - Disables storing of future hints on the current node.

Usage – nod**etool disablehandoff**

DisableHintsfordc - Disable hints for a datacenter.

Usage – **nodetool disablehintsfordc <datacenterName>**

DisableThrift - Disables the Thrift server

Usage – **nodetool disablethrift**

Drain - Flushes all memtables from the node to SSTables on disk. Cassandra stops listening for connections from the client and other nodes. You need to restart Cassandra after running **nodetool drain.** You typically use this command before upgrading a node to a new version of Cassandra.

Cassandra Adminstration components

NodeTool

EnableautoCompaction - Enables autocompaction for a keyspace and one or more tables.

Usage – **nodetool enableautocompaction <keyspace>**

Enablebackup - Enables incremental backup.

Usage – **nodetool enablebackup**

Enablebinary - Re-enables native transport.

Usage – **nodetool enablebinary**

Enablegossip - Re-enables gossip

Usage – **nodetool enablegossip**

Cassandra Adminstration components

NodeTool

Enablehandoff - Re-enables the storing of future hints on the current node.

Usage – **nodetool enablehandoff**

We can also use pausehandoff to pause the hints.

Usage – **nodetool pausehandoff**

You can resume Hinted Handoff by resumehandoff command

Enablehintsfordc - Enable hints for a datacenter.

Usage – **nodetool enablehintsfordc <datacenter_Name>**

Enablethrift - Re-enables the Thrift server.

Usage – **nodetool enablethrift**

Cassandra Adminstration components

NodeTool

Failuredetector - Shows the failure detector information for the cluster.

```
[root@cassandra-1 ~]# nodetool failuredetector
  Endpoint,                  Phi
/172.31.93.26,          0.43395649
/172.31.93.245,          0.35259417
```

Flush - Flushes one or more tables from the memtable.

Usage – **nodetool flush <keyspace/table>**

Gcstats - Print garbage collection (GC) statistics.

```
[root@cassandra-1 ~]# nodetool gcstats
       Interval (ms) Max GC Elapsed (ms)Total GC Elapsed (ms)Stdev GC Elapsed (ms)   GC Reclaimed (MB)         Collections      Di
rect Memory Bytes
            29640339                364                6852                  31          3640707976                  22
             -1
[root@cassandra-1 ~]#
```

Cassandra Adminstration components

NodeTool

Getcompactionthreshold - Provides the minimum and maximum compaction thresholds in megabytes for a table.

```
[root@cassandra-1 ~]# nodetool getcompactionthreshold navaneeth_ks bank_details
Current compaction thresholds for navaneeth_ks/bank_details:
 min = 4,  max = 32
```

Getcompactionthroughput - Print the throughput cap (in MB/s) for compaction in the system.

```
[root@cassandra-1 ~]# nodetool getcompactionthroughput
Current compaction throughput: 16 MB/s
[root@cassandra-1 ~]#
```

Cassandra Adminstration components

NodeTool

Getendpoints - Provides the IP addresses or names of replicas that own the partition key.

Usage – **nodetool getendpoints <keyspace> <table> <key>**

Getlogginglevels - Get the runtime logging levels.

Usage - **nodetool getlogginglevels**

Getsstables - Provides the SSTables that own the partition key.

Usage – **nodetool getsstables <keyspace> <table> <partition_key>**

Getstreamthroughput

Usage – **nodetool getstreamthroughput**

Cassandra Adminstration components

NodeTool

Gossipinfo - Provides the gossip information for the cluster.

Usage – **nodetool gossipinfo**

Info - Provides node information, such as load and uptime.

```
[root@cassandra-1 ~]# nodetool info
ID                     : ecf730ec-0e0c-4876-b652-40773ae88e07
Gossip active          : true
Thrift active          : true
Native Transport active: true
Load                   : 769.63 KB
Generation No          : 1602083739
Uptime (seconds)       : 32190
Heap Memory (MB)       : 260.75 / 1934.00
Off Heap Memory (MB)   : 0.01
Data Center            : dc1
Rack                   : rack1
Exceptions             : 0
Key Cache              : entries 32, size 2.86 KB, capacity 96 MB, 92 hits, 131 requests, 0.702 recent hit rate, 14400 save period
 in seconds
Row Cache              : entries 0, size 0 bytes, capacity 0 bytes, 0 hits, 0 requests, NaN recent hit rate, 0 save period in seco
nds
Counter Cache          : entries 0, size 0 bytes, capacity 48 MB, 0 hits, 0 requests, NaN recent hit rate, 7200 save period in sec
onds
Token                  : (invoke with -T/--tokens to see all 256 tokens)
```

Cassandra Adminstration components

NodeTool

Invalidatecountercache - The **nodetool invalidatecountercache** command will invalidate the counter cache, and the system will start saving all counter keys.

Invalidatekeycache - Resets the global key cache parameter to the default, which saves all keys.
By default the key_cache_keys_to_save is disabled in the cassandra.yaml. This command resets the parameter to the default.
Usage – **nodetool invalidatecache**

Invalidaterowcache - Resets the global key cache parameter, row_cache_keys_to_save, to the default (not set), which saves all keys
Usage – **nodetool invalidaterowcache**

Cassandra Adminstration components

NodeTool

Join – makes the node to join the ring, if it was not joined at first due to some issue.

Usage – **nodetool join**

Listsnapshots - Lists snapshot names, size on disk, and true size.

Usage – **nodetool listsnapshots**

Mark_unrepaired - Mark all SSTables of a table or keyspace as unrepaired. Use when no longer running incremental repair on a table or keyspace.

WARNING: This operation marks all targeted SSTables as unrepaired, potentially creating new compaction tasks. Only use if no longer running incremental repair on this node.
Usage – **nodetool mark_unrepaired –f keyspace**

Cassandra Adminstration components

NodeTool

Netstats - Provides network information about the host.

```
[root@cassandra-1 ~]# nodetool netstats
Mode: NORMAL
Not sending any streams.
Read Repair Statistics:
Attempted: 0
Mismatch (Blocking): 0
Mismatch (Background): 0
Pool Name                    Active   Pending      Completed   Dropped
Large messages                  n/a         0              0         0
Small messages                  n/a         0             22         0
Gossip messages                 n/a         0          98104        39
[root@cassandra-1 ~]#
```

Proxyhostograms - Provides a histogram of network statistics at the time of the command.

```
[root@cassandra-1 ~]# nodetool proxyhistograms
proxy histograms
Percentile      Read Latency     Write Latency     Range Latency
                    (micros)          (micros)          (micros)
50%                    0.00              0.00              0.00
75%                    0.00              0.00              0.00
95%                    0.00              0.00              0.00
98%                    0.00              0.00              0.00
99%                    0.00              0.00              0.00
Min                    0.00              0.00              0.00
Max                    0.00              0.00              0.00
```

Cassandra Adminstration components

NodeTool

Rebuild - Rebuilds data by streaming from other nodes. Similar to bootstrap but for the data. This will table more time when data re huge in cluster and make more IO

Usage – nodetool rebuild –m normal

Modes

normal - conventional behavior. Streams only ranges that are not locally available. Default.
refetch - resets locally available ranges. Streams all ranges but leaves current data untouched.
reset - resets locally available ranges. Removes all locally present data (like a TRUNCATE). Streams all ranges.
reset-no-snapshot - resets locally available ranges. Removes all locally present data (like a TRUNCATE). Streams all ranges. Prevents a snapshot if auto_snapshot is enabled in cassandra.yaml.

Rebuild_index - Performs a full rebuild of the index for a table
Usage – **nodetool rebuild_Index**

Cassandra Adminstration components

NodeTool

Refresh - Loads newly placed SSTables onto the system without a restart.

Usage – **nodetool refresh keyspace table**

RefreshsizeEstimates - Refreshes system.size_estimates table. Use when huge amounts of data are inserted or truncated which can result in size estimates becoming incorrect.

Usage – **nodetool refreshsizeestimates**

Removenode - This command removes a node, shows the status of a removal operation, or forces the completion of a pending removal. When the node is down and nodetool decommission cannot be used, use nodetool removenode. Run this command only on nodes that are down.

Usage – **nodetool removenode <host_id>**

Cassandra Adminstration components

NodeTool

Repair - The repair command repairs one or more nodes in a cluster, and provides options for restricting repair to a set of nodes. Performing an anti-entropy node repair on a regular basis is important, especially in an environment that deletes data frequently.

Usage – **nodetool repair**

Replaybatchlog - Replay batchlog and wait for finish.

Usage – **nodetool replaybatchlog**

Ring - Provides node status and information about the ring.

Usage – **nodetool ring**

Cassandra Adminstration components

NodeTool

Stop - Stops the compaction process

Supported types are COMPACTION, VALIDATION, CLEANUP, SCRUB, VERIFY, INDEX_BUILD.

Usage – **nodetool stop cleanup**

Stopdaemon - Stops the cassandra daemon.

Usage – **nodetool stopdaemon**

Tablehistograms - Provides statistics about a table that could be used to plot a frequency function

Usage - **nodetool tablehistograms navaneeth_ks bank_details**

Cassandra Adminstration components

NodeTool

Toppartitions - Samples database reads and writes and reports the most active partitions in a specified table.

Usage - **nodetool toppartitions navaneeth_ks bank_details 10000**

Tpstats - Provides usage statistics of thread pools.

```
[root@cassandra-1 ~]# nodetool tpstats
Pool Name                    Active   Pending      Completed   Blocked  All time blocked
MutationStage                     0         0           4924         0                 0
ViewMutationStage                 0         0              0         0                 0
ReadStage                         0         0             25         0                 0
RequestResponseStage              0         0           9764         0                 0
ReadRepairStage                   0         0              0         0                 0
CounterMutationStage              0         0              0         0                 0
MiscStage                         0         0              0         0                 0
CompactionExecutor                0         0          21220         0                 0
MemtableReclaimMemory             0         0             50         0                 0
PendingRangeCalculator            0         0              3         0                 0
GossipStage                       0         0         103405         0                 0
SecondaryIndexManagement          0         0              0         0                 0
HintsDispatcher                   0         0              0         0                 0
MigrationStage                    0         0              1         0                 0
MemtablePostFlush                 0         0             70         0                 0
ValidationExecutor                0         0              9         0                 0
Sampler                           0         0              0         0                 0
MemtableFlushWriter               0         0             50         0                 0
InternalResponseStage             0         0             26         0                 0
AntiEntropyStage                  0         0             41         0                 0
CacheCleanupExecutor              0         0              0         0                 0
Native-Transport-Requests         0         0            554         0                 0

Message type           Dropped
READ                         0
RANGE_SLICE                  0
_TRACE                       0
HINT                         0
MUTATION                     0
COUNTER_MUTATION             0
BATCH_STORE                  0
BATCH_REMOVE                 0
REQUEST_RESPONSE             0
PAGED_RANGE                  0
READ_REPAIR                  0
```

Cassandra Adminstration components

NodeTool

Truncatehints - Truncates all hints on the local node, or truncates hints for the one or more endpoints.

Usage – **nodetool truncatehints <ipdetails>**

Upgradesstables - Rewrites SSTables on a node that are incompatible with the current version. Use this command when upgrading your server or changing compression options.
Usage – **nodetool upgradesstables keyspace table**

Verify - Verify (check data checksum for) one or more tables.
Usage – **nodetool verify navaneeth_ks bank_details**

Cassandra Adminstration components

Cassandra Stress Tool

- The cassandra-stress tool is a Java-based stress testing utility for basic benchmarking and load testing a Cassandra cluster.

- Data modeling choices can greatly affect application performance. Significant load testing over several trials is the best method for discovering issues with a particular data model.

- The cassandra-stress tool is an effective tool for populating a cluster and stress testing CQL tables and queries. Use cassandra-stress to:

  Quickly determine how a schema performs.
  Understand how your database scales.
  Optimize your data model and settings.
  Determine production capacity.

Cassandra Adminstration components

Cassandra Stress Tool

Write – Multiple concurrent reads. The cluster must first be populated by a write test.

cassandra-stress write n=1000000 -rate threads=50 -node cassandra-1

Read - Multiple concurrent writes against the cluster.
cassandra-stress read n=200000 -rate threads=50

Cassandra Adminstration components

Good Practices

1. Get your partitions right

- Uneven partitions is one of the biggest reasons for non-deterministic reads/writes performance and query timeouts. It also causes uneven pressure on compaction cycles as some partitions may be big and may take long time to compact while smaller partitions may compact quickly.

2. Proper Indexing, especially secondary Index

3. Locate only Cassandra daemons in the node, sometimes keeping SOLR or elasticsearch in Cassandra creates resources share

4. Avoid Full Read/Writes

- Mutations incur a heavy cost on reads as reads have to consult multiple *SSTables* to ensure that last update is always returned. Mutations are compacted / discarded during the compaction process, however that causes significant load on the cluster if the real content is not changing.

Thanks