NoSQL – Cassandra:

Why is there a need for NoSQL technology

Size of data ranges from: Small, Medium and Large.

To process a small text file we can write a Java program to read a file, search for a sentence in it etc.,

When it comes to medium sized data we use technologies like: MySQL which allows concurrency.

But when the size of data becomes extremely large and the system uses Relational technologies.

There are several issues that might arise:

1. Consistency is a fallacy, because RDBMS uses Master-Slave architecture. When a master writes to a slave, there is a read lag that occurs when the users tries to read from the slave. Hence there are chances of the user reading old data rather than the latest information, leading to loss of consistency.
2. Sharding, is when we divide the data to be distributed among several databases for example, we can divide the population database into 50 state based separate shards/databases. The problem with this is, when a query needs to be answered. All 50 databases need to be queried to find the answer which causes an enormous lag. Also while moving the data from one central system to individual databases the script that does this job needs to be tested perfectly and needs constant monitoring.
3. Availability is a paradox, in a distributed system even though the data is stored in multiple data centers. When the system goes down, higher availability makes sure the data is not lost completely. But there is always the hours and hours of time that is spent on “rebooting” the system. So when we say system is Always Available it is not completely true.
4. Scaling is a nightmare, scaling using Relational systems often ends up expensive, Sharding data creates consistency & lags in response time, we have to keep updating the application frequently based on how the data is being “scaled to size” to improve response times. Queries change with the way data scaling happens, leading to the focus being on maintenance and scaling rather than innovating and adding new ideas to the product. Imagine de-normalizing a huge database, it is an absolute nightmare.

Hence we can learn that Consistency is not absolute, so instead of being absolute we manage it.

Instead of Sharding manually, we make it a built in feature which the system manages, hence when we scale from 3 databases to 30, we rarely notice any issues.

Master-Slave architecture creates consistency issues which worsen with scaling and Sharding, hence we remove the Master-Slave architecture.

Scaling up using technologies such as SAN is highly expensive, hence we build a system that can scale using commodity hardware. Horizontal Scaling, adding more machines to existing system than using expensive Network storage and Vertical scaling. Hence keeping the costs low.

Rather than using Scatter-Gather queries or expensive joins spread over multiple shards, we build a data model that automatically directs us to a single machine than having to search all databases/shards in the system. Thereby reducing query time and improving performance with scaling.

What is Apache Cassandra

Apache Cassandra is a Fast, Distributed and De-Centralized Database built for high availability and linear scalability. That means we can predict performance and guarantee an SLA (Service level Agreement), that means be it 3 nodes or 300 nodes the performance doesn’t degrade with little latency.

There is no Single Point of Failure (SPOF), because Cassandra is de-centralized distributed database, it does not have a Master-Slave architecture, nor does it have the election to choose a master database. Out of the box the system can scale from a single system to multiple data centers without a single point of failure because there is no single node that holds control/authority. If a node fails another node/data center can take up the service immediately with little latency and there is next to no reboot time, only in special cases.

Cassandra can be deployed in Commodity hardware and is cheap. Easy to deploy and manage with small teams.

Not a drop in replacement for RDBMS, Cassandra has a specific data model which makes it confirm to certain norms which allow the designer the flexibility in scaling and managing the system.

Cassandra can be seen as a “Giant Hash Ring”, where each node confirms to an individual commodity server and all these commodity servers/nodes work together work as a “Ring”. Each node owns a range of hashes.

When we create the Data Model we create a Primary Key. A part of this primary key is a “Partition Key”. When the query executes, the Partition Key goes through a Hash Function which is associated with a node in the Ring. Hence any query which contains the same partition key will go to a specific node and each node owns a range of hashes. Thereby a range of queries whose partition keys fall within the range of a node, they will be directed to the node that “owns” that range.

Cassandra also replicates this data that belongs to a single node by design to other nodes, so that when a node fails the failed node can be replaced but the users experience no lag. Because, the nodes that share the data continue to serve the requests often with no observable lags.

CAP Theorem: During a network partition either between data-centers or on the same network you can either choose availability of consistency. The fact is during a network partition there can be no consistency. When we want to be completely consistent during a network partition the system cannot talk to each other and it would appear that the communication is down. So if we chose to be highly available than communicate with another system to be consistent, the system can be highly available and that’s what Cassandra choses.

When we have data distributed across data centers, it is very tough to be consistent because we are bound by the speed of light. It takes time for data to move from one data center in USA to another in India and varies upon network capacity as well. Hence if we choose consistency here issues arise as query might arrive faster than the heavier most updated data. Hence, we choose availability over consistency. We choose to replicate data asynchronously from data center to data center because it takes a much longer time for data to be shared between data centers. This is something we are limited by the speed of light.

Cassandra gives the control to the designer in the sense that you can tune the amount of consistency that is needed.

**Replication Factor:** Cassandra replicates data, the general Replication Factor being 3. Which means for every write tat is done, 3 copies of the same write are stored across 3 nodes, one node that directly corresponds to the hash value of the partition key and the other two nodes that hold the replicated range of hash values. We set the replication while creating the “Keyspace”, which is a bunch of related tables. Similar to a Schema in Oracle and a database in MySQL. This replication happens asynchronously.

While the replication happens if a machine/node goes down, the node that we are talking to stores a “HINT”. When the node comes back up to join the Cluster/Ring, Cassandra replays the “HINT” and writes all the data from the time of the HINT to bring the node back to consistency of other nodes and to start servicing allowing the load to be shared, reducing the stress of the total network.

**Consistency Level:** Consistency levels are set at the query level. Consistency level means, how many replicas does the system needs to listen from in order for the read or write to be considered successful. For example: How many times does the system needs to get the confirmation in order for Cassandra to acknowledge to the user that the WRITE was successful. How many times does the system needs to acknowledge in order for data to be read back to the user when a READ operation occurs. The replication factor will be honored irrespective of Consistency level.

Two most popular levels of Consistency to hear back from the system are: ONE and Quorum. Irrespective of these Consistency Levels Replication Factor will be honored and data will be replicated asynchronously. Quorum means majority of the nodes should agree, hence when the Replication Factor is 3 Quorum would mean at least 2 nodes should respond, in order for the read or write to be acknowledged as successful. The advantage that these have is how fast you can reply and how low you can tune the latency to, thereby contributing to execution speed of the application overall.

This effects Availability, in the sense when RDBMS say they choose consistency over availability, they mean they will wait for every node that the data is replicated to, to respond/acknowledge the READ or WRITE. Cassandra allows the designer to choose, in cases where you need it you might choose the replication factor and consistency to be the same thereby making system consistent for that specific use case, for others scenarios within the same application that might change. Also for example, if the designer chooses Consistency Level Quorum out of 3 nodes of replication factor, even if one node goes down he will still be able to keep the system going similar is the case with Consistency Level One. We should remember that these nodes might exist on the same data center or across data centers. Hence you can tune consistency “Per-Query”. This allows for much higher availability even when a whole data center goes down.

Another important distinction to make is Data Centers can be logical. We can create two logical data centers with different consistency levels and use one for OLTP and another for OLAP. This can allow queries to be focused, targeted, available, consistent and faster.

HOW CASSANDRA READS and WRITES WORK

In Cassandra there are no Master-Slaves like in RDBMS, meaning any node can start servicing your request. The node that starts to service the request is called “Coordinator” node. This is not necessarily the node that contains the partitioned & hashed data, it can be any node. It is called Coordinator node because it coordinates with rest of the nodes that contain the data. Any node can service any request, meaning the Coordinator node can change from request to request be it READ or WRITE.

When a WRITE requests comes in 2 things happen:

1. The data is written to a “append-only” data structure called a “COMMIT LOG”. Since this is going to be a Sequential IO it is going to be very fast even disks.
2. Once Cassandra gets that durability, it then merges that data into an “in-memory” representation called a “MEM-TABLE”.

Then Cassandra acknowledges to the Coordinator node that the write is done, the acknowledgement is successful if the Consistency level is met and the Replication Factor is achieved asynchronously. It waits on the Consistency levels not on the replication factor, the system does not wait for replication factor (isn’t synchronous) but it waits for the consistency level to be met.

Because the WRITEs are so fast, Cassandra is often referred to as WRITE-OPTIMIZED database. It can write lots of data very fast and one of the reasons adding to the speed is the write path and the commit log being sequential IO.

MEM-TABLE is an in-memory representation/data structure which is limited by the size of memory. Hence Cassandra flushes it to the disk asynchronously, again a whole bunch of sequential IO written to the disk in the form of SS Tables. It basically takes the IN-MEMORY representation of MEM-TABLE and serializing it to disk. In Cassandra every write to a Column contains a TIMESTAMP. Cassandra also does not do UPDATE or DELETE in place. The Commit Log or SS Tables are IMMUTABLE, meaning they can’t be updated, once written they cannot be changed. Cassandra maintains a special kind of value called a TOMBSTONE with a TIMESTAMP. Which means as of this time and date the column has no record, which allows for the system to be Serializable and have Sequential IO.

As and when the MEM-TABLES/Memory gets Full. They are written to the disk and space is made for recent data to be held for read or write. From time to time the MEMTABLE is flushed asynchronously to the disk as an SS Table with each column having a time stamp and a special value called TOMBSTONE.

Even these SS Tables can become many over time. To reduce the count, a process called COMPACTION happens. Over time smaller SS tables are taken into-memory, the row with the latest timestamp is considered as latest and the row with older time stamp is ignored not deleted.

When we read data, there are chances that compaction hasn’t occurred on all SS Tables and some of the data might still be in-memory, yet to be written on to the SS tables. Hence when a READ request arrives data from multiple SS Tables is pulled into memory, merged together with the latest time stamp being the winner, if the latest data is still in the MEM-TABLE yet to be written into the SS table, the data is merged with the latest time stamp and returned to the coordinator node, which returns the data to the user.

When we do a READ or WRITE heavy workload the type of disk used might alter the response time because of processes like COMPACTION which might eat away some time.

When we use a consistency level of less than ALL, where all nodes that the data is replicated to do not need to acknowledge. Cassandra does something called a “READ-REPAIR”. There is chance that from time to time nodes might disagree on what is the latest data that each has. Hence Cassandra with a default of 10% of all READ time, tries to coordinate with all nodes with replicas and makes sure the data is consistent all across the nodes with the replicated information. That is why Cassandra is called an “Eventually Consistent” data system.

Two distributions:

1. Apache Cassandra Open Source: A community maintained latest and most updated resource, it is also possible to have errors from recent or old updates that need to be fixed yet. JIRA’s can be raised and fixed through open communication with the community members.
2. DataStax Enterprise: Commercial distribution, open source Cassandra at core. A bit risk averse with additional QA with dedicated support. Also contains cool features such as: Multi Data center Cassandra and Multi Data Center search using Lucene/SOLR Queries in near real-time with OLTP system. Also get integrated Spark or Hadoop for analytical queries. We get collocated SPARK nodes. For startups DataStax Enterprise program allows to be given for free.

Commands in CQL:

1. To start DSE Cassandra: ./dse Cassandra
2. To launch CQL shell: cqlsh
3. To get status if the node is up or down: nodetool status, tells if node is up or down or leaving etc., along with node address, load and the data center the node occupies.
4. Create Keyspace:

Create Keyspace “Keyspace\_name” with replication={‘class’:’SimpleStrategy’,’replication\_factor’:1}

1. To create Table:

Create table videos(video\_id timeuuid, date\_added timestamp, title text, PRIMARY KEY(video\_id));

\*\* UUID’s enable several nodes to generate non clashing id values. This is the way Cassandra makes sure nodes do not have the same id values without actually comparing it using UUIDs. Time UUID is a UUID with an Embedded timestamp value. This can be used to produce time ordered data for example: Credit Card usage. We can use CQL’s extract time function to extract the time part of the UUID.

1. To copy an entire CSV file:

COPY videos(video\_id, added\_date,title) from ‘file\_path/filename.csv’ with HEADER=TRUE;

Other tools available to load data into Cassandra are: Apache SPARK and DataStax Bulk Loader.

PARTITIONS

Partition Key decides where in the Cassandra Ring the values are stored. For every partition key value, a corresponding Hash value is computed and each node in the ring is made responsible for a certain range of these hashes. Hence when the Hash value is computed for value in column, they can be assigned to or be identified as belonging to a certain token range and get assigned to the node responsible. This is a “Constant Time Complexity” event.

The 1st column in the Composite Primary Key set is always the Partition Key. In the “videos” table “video\_id” is both the partition key and the primary key. In real time it might not be enough for a single column to be part of the Primary Key, where the Primary Key itself is the partition key column and there are no other column. This is riskier. What we generally do in the Primary Key along with Partition Key Column, we add few more columns called “Clustering Columns”, together they create a unique record. Where the Partition Key helps to identify the node which the data rests on in constant time, the remaining part of the primary key i.e., the Clustering Columns satisfy the need to create a pattern that needs to be followed to insert and retrieve records in a super fast manner. For example: Diving federal data into state based nodes where State Name is the Partition Key and probably an SSN or Driving License can be used as Clustering Column in increasing or decreasing order. Since Primary Key Columns are indexed retrieving this data happens super fast.

The query to see the partitioned token value is select token(video\_id),video\_id from videos;

When we query non partitioned & non-primary key columns the system rejects it. For that to happen the user needs to create secondary indexes or use “ALLOW FILTERING”. This will reduce performance and increase search time. Which defeats the purpose of Cassandra.

In Cassandra tables are designed based on the way user queries might happen. Which helps the designer to create super fast insert and read queries.

Partitions group rows physically together based on partition keys.

Partitioner hashes the Partition Key value to generate a partition token, which is then identified with a node which holds responsibility for a range of such rows. All these partitioned/hashed/token values which belong to a node are physically held together on disk, enabling faster read and writes.

A well chosen combination of Partition Key followed by a better Clustering Columns makes all the difference in the Cassandra Data Model. A well modeled Composite Primary Key with a unique Partition Key Column and related Clustering Columns make all the difference.

A well written combination of Partition Key and Clustering Columns perform a similar action in RDBMS called GROUP BY clause, except that now it is part of the model and queries are defined. It is faster because it does not need to do any aggregation and spend anymore memory or time.

The Partition Key helps locate the “node” with partition token values in Constant Time Complexity and the Clustering Column not only ensured the records are unique as is the job of the primary key, but when ordered in ascending or descending order. They can be used in applications such as: Help detect Credit Card Fraud.

For example: If State is the partition key for a federal program, city, county and pin code can be used as Clustering Columns to group related data together. We can defined or think of clustering columns as basis of the “SORT ORDER” or GROUP Order. Where the entire group has the same Partition Key but is grouped by City, County and Pin Code, now there might still be people with same name. So we could add user\_id with column type UUID or Time UUID, which makes sure every record that is generated is unique. Hence when we add “ID” to the Clustering Columns it makes a lot of sense.

To be noted Cassandra does not allow us to use “alter Table” command to change primary key. Hence the data model that is used should be well thought of on how the query will be and modeled accordingly. If we have to change the query structure of the primary key. We will have to re do the entire data model. Primary Key can be composite with Partition Key and Clustering Column. Partition Key itself can contain more than one column. Hence it is a better practice to use () around partition key even if there is just a single column or more than one column in the partition key.

When querying Cassandra, using Partition Key column is a must.

Then followed in the order of Clustering columns.

Especially when performing an “EQUALS” query. If the order is changed an ERROR will be shown.

The way the Partition Key and Clustering Columns are set. The search is done as a Binary Search and that is super fast.

While using Clustering Columns, Cassandra allows each Clustering Column to be ordered in a different fashion such as:

PRIMARY KEY ((state),city,name,id) WITH CLUSTERING ORDER BY(city desc, name asc);

For example when fires in california happen, we can build a system using an individuals phone to track their location and note into the system every minute. So if they get lost the fire department knows where to focus .

Apache Cassandra NODE

Cassandra is built on the back of individual nodes, together these nodes form a Cluster or a Ring.

A Node is a Java Process, run on the back of a JVM. Cassandra can run on Cloud, on-premise data center or an individual disk.

A Node is responsible for the data that is stored on it. All the data on the Node is stored on a “Distributed Hash Table”.

An individual Node in Cassandra can handle up to 6000-12000 transactions/sec/core. It can manage data of size up to 4 Terabytes.

If the server/system on which the Node is running has multiple cores, the system will then be able to manage a lot of transactions.

An individual Node can be managed using a tool aptly named: Nodetool, “nodetool status” provides information about the properties of the node. It has some node specific tools to check on the node and some cluster specific tools that help monitor the health of the cluster as well.

For example: nodetool info gives information such as JVM statistics.

Node Tool Commands:

1. ./nodetool help – Lists the entire nodetool commands that can be used to manage a node and get information about the cluster as well.
2. ./dsetool info - Can be used with Cassandra, Apache Spark, Apache Solr and Graph
3. ./nodetool status gives details of the cluster, the snitch, the IP address, total percentage of data stored on each node, the state of each node and information of each node.
4. ./nodetool info – Details information about an individual node that the user is connected to.
5. ./nodetool describecluster – Shows settings that are common across all nodes in the cluster and current schema version used by each node.
6. ./nodetool setlogginglevel – Allows to dynamically change the logging levels of Apache Cassandra without the need to restart.
7. ./nodetool settraceprobability 0.1, a decimal value with a range between 0 and 1, representing percentage of queries being saved with 1 being 100%.
8. ./nodetool drain, stop writes from occurring on the node and flushes all the data to the disk.
9. ./nodetool stopdaemon – Stops the current node from running.
10. ./dse Cassandra – Command to start and launch Cassandra
11. ./Cassandra-stress write n=50000 no-warmup -rate threads=1 , populates the cluster with 50000 partitions, using 1 client thread. Shows partition rate, ops/second etc.,
12. ./nodetool flush, unlike nodetool drain, flush allows to data to be written to the node and commits all data to disk.

Apache Cassandra Ring/Cluster

We can continue using a single node and continue adding expensive storage and memory. But why do that says Cassandra. Cassandra’s Ring/Cluster is a group of nodes working together as a Cluster/Ring with no Master/Central control.

Each NODE in Cassandra is responsible for a range of Hashes and these hashes are asynchronously replicated to 2 other nodes if the replication factor is 3. Hence when the request comes in, any node can start servicing the request. This allows any node from the POOL of available nodes can service the request, instead of election a Master. Cassandra is a Query based data model, hence when a query is created to insert or retrieve data from a node. The first column should be that of the PARTITION KEY, which is hashed to a unique value in Constant time complexity and finds the node that can service the request. Hence when a request comes in, any node that is available can pick up and start servicing the request that is why such nodes are called COORDINATOR nodes. Once the COORDINATOR nodes picks up the request, it then passes the PARTITION KEY to be hashed to identify node that contains the Hash and request is sent to that node. If that node is down, a HINTED HANDOFF is stored and the servicing responsibility is given to nodes which duplicated the data earlier. Once the node comes back, it then plays the hinted handoffs to get to the most updated records. Then it starts servicing. This makes sure the Cassandra system is highly available, much more than an RDBMS.

Look at the Cluster/Ring like a system of volunteers who share similar thoughts and open to help each other. They believe in growing/scaling by agreeing to help each other. An organic growth if you may call it. It provides anti-failure capability and continues to service a request lets say: Netflix using Cassandra can continue playing a video through another node which has the duplicate data without next to no lag. Because of the Cluster Cassandra enjoys the ability to be highly available and consistency even though RDBMS systems claim to be, is to be realized to be a paradox. It is like a council of ministers without a prime minister where a resolution is adopted if majority agrees, which is what we set using CONSISTENCY LEVELS.

In terms of Cost the Cluster allows the nodes to grow by attaching more commodity hardware, rather than expensive network attached storage solutions and it can be scaled to multiple data centers right out of the box.

A single NODE by itself can handle up to 12000 transactions/second/core but we all know the stress real world systems face. This is not enough, the CLUSTER/RING shared the stress and you do that by connecting individual systems/commodity hardware through Horizontal Scaling than adding expensive network storage solutions such as SAN to existing machine to share the load. Just a reminder that Commit Log is Append Only DS so Write or Read is done extremely fast.

Data that can be stored lies between –(2^63) to +((2^63)-1), that large enough storage to manage all your data on a 64-bit compute storage system.

PARTITIONER

The Partitioner determines which data resides where. It decides which token value, hashed from the partition key belongs to which part of the ring. This is again based on the token range each node shares and the Partitioner keeps knowledge of this range to distribute data coming in to and from the node.

It is such a critical part of the job, that if the Partitioner goes wrong few nodes might end up getting loaded with all data. Thereby breaking the essential aspect of load sharing and balancing act of the Cluster/Ring.

It is better to use Partitioners such as: MURMUR3 and MD5 which spread data randomly but evenly all over the Cluster/Ring. It keeps in view the Hash Range of the nodes in the cluster while doing this.

Some kinds of NODE types to remember on the Cluster:

1. Coordinator – Any node can be a Coordinator Node. Because Cassandra does not have Master Node. Any Node from the pool of Nodes that are not servicing a request or can handle the load, can take up the job of a coordinator node. Its job is to: Receive the request, Hash the value, identify which node the token belongs to and direct it.
2. SEED Node & New Node – Every node has 4 stages, UP-DOWN-Joining-Leaving. While joining the Cluster a new node communicates with SEED nodes, which share information about the Cluster Topology and information. So that load can be evenly balanced, and the system can scale, and a node can join while the system is online. No downtime or reboot time. SEED nodes can be configured in “cassandra.yaml” file.

DRIVERS

While communicating with Cassandra NoSQL DB, we do it via Drivers built for each language example: Cassandra-Java driver. The advantage that this Driver has is, it participates in the communication. Once the driver connects to Cassandra Cluster to “Pass Queries and Receive replies”, it can store some of that data such as: Token Range, nodes in the cluster and their status. This will help the driver to direct the requests directly to the node which holds the data or nodes that hold the replica, instead of using the Coordinator Node.

Some of the policies of the driver are:

1. TokenAwarePolicy, which is self-descriptive.
2. RoundRobinPolicy, driver Round Robins the Cluster.
3. DCAwareRoundRobinPolicy, driver Round Robins data to be within a Data Center.

Apache Cassandra is topology aware, that’s how NODETOOL operates on awareness. Hence the drivers also should.

How does this related to scaling? A single server/node cannot handle something a Cluster in Cassandra can. Hence when a driver is aware of the topology of the cluster and instead of bringing in an expensive machine, we just add more commodity hardware to existing cluster and create a new node to join the cluster. Hence you do not need to downtime the node and transfer the data. The cluster handles that while the system stays online and scales. When the driver gets to know the topology it can save time and increase execution speed by knowing which node holds which token range and direct the request accordingly. This does not mean COORDINATOR nodes are obsolete. This is an added help to COORDINATOR node to save load and improve execution speed thereby adding to scaling.

To sum up as we keep adding more nodes we add the capacity, share load while system remains online. Thereby remaining truly highly available and scaling together which is rare even with other NoSQL Databases.

HORIZONTAL SCALING VS VERTICAL SCALING:

Vertical Scaling requires one large expensive machine, even if it is a SAN. The entire load ends up being on a single network.

Horizontal scaling refers to adding less expensive and at the same time capable nodes to the cluster, which then share the load without putting it on a single machine or a single network. Plus having PARTITION KEY helps identify Nodes that can service request in Constant Time O(1). Which when compared to SAN, RDBMS is enormously fast.

If two nodes can handle 100,000 transactions/sec. Adding 2 more i.e., total 4 nodes will increase the load balancing/scaling capacity to 200,000/sec and 8 nodes to 400,000 transactions/sec. Doubling nodes doubles capacity. Cassandra has ability to scale as you need to without having to hold user requests from being serviced during downtime. One issue is to note is, more nodes can be added to scale during busy days like: ThanksGiving and once done. The additional nodes can be DOWNED because the load would be down by a large volume after that day. Then can be used again when necessary.

**Commands** to create a 2 node cluster:

1. Extract tar file with DSE Cassandra into directory called node1,

Then execute the command labwork/config\_node 1

1. Repeat step 1 now name the folder node 2 and run the command ./config\_node 2.

Now that node 1 and node 2 are configured one thing to change will be INITIAL\_TOKEN value in node 2 located in /node2/resources/Cassandra/conf/Cassandra.yaml.

1. Open Cassandra.yaml file to change INITIAL\_TOKEN value to 9223372036854775807. This indicates the second node
2. From node1 directory start the first node as: ./dse Cassandra,

once the node starts click ENTER and do the same at node2 ./dse Cassandra.

Observe the second node 2 takes more time to bootstrap and join the cluster.

Though both these nodes are being created on a single machine for development.

This is an example of how Cassandra can be configured out of box to join the cluster and through this process the SEED nodes will stream data and topology to the new node.

1. Once both the nodes are up, run the command ./nodetool status.

You should see two nodes each with status “UN” – Up & Normal, their rack, the IP, the data center should be CASSANDRA.

1. Now run the command ./nodetool status,

Now you should be able to see two nodes each with its IP, Rack that it is on and the Token Range it shares. Now that we have done this, we can create multiple such nodes for learning how to create a cluster.

1. Create the tables videos and videos\_by\_tag,

Now execute select token(tag),tag from videos\_by\_tag

This shows which token\_value/hash exists on which node of the Cluster.

1. Now run the command: ./nodetool ring

This shows how each node contains a range of tokens.

If we keep adding 2 more nodes to this. It will show 2 more nodes in the Cluster.

1. ./nodetool getendpoints killrvideo videos\_by\_tag ‘cassandra’

Shows the IP address on which the node ‘cassadra’ exists.

Same for ‘datastax’.

(SHARDING + RDBMS) vs Peer-to-Peer

In an RDBMS to scale the system we “Shard”, Sharding is a process of dividing the large data in the keyspace or DB among multiple smaller data parts called Shards that run on different machines. This is called Horizontal Partition.

Generally we have a Master/Leader who manages requests and maintains Followers to handle loads. Replicas of the Leader-Follower are maintained in different data centers, to make sure if one data center goes down the data is not completely lost. The leader node continue to service requests and the other replicas can be used to maintain redundancy and use them for OLAP, while the main Leader continues to service OLTP requests.

Imagine instead of using the replicas for OLAP, they are used for Sharding/Horizontal Partitioning. Data from the main node is shared among all other nodes to reduce load on a single node. This might help manage demand a little. But the issue that would arise now is:

How would one know which data is part of which SHARD? Sharing data across nodes in RDBMS system requires us to build a tool which can remember which Shard contains which Data and possibly in which range. This also will bring in much more cost of time/latency to service requests when a JOIN or AGGREGATION needs to be done.

Another important issue that might arise is, assume data is Sharded among multiple nodes in an RDBMS. Now the node that has the Horizontal Partition is servicing the request. What happens if that node goes down? We have now to wait for that node to fail-over and restart. This also keeps other nodes waiting for the aggregation to complete. Since for aggregation Synchronization is Crucial, the system and the user have to wait for long amounts of time. Time is money.

Earlier discussed is the Split-Brain Problem, is when Leader & other Followers are down or are not visible to other followers because of network issues. The remaining Follower nodes will have to now re-elect a Leader. Now what if the nodes that went down come back online? Now we will have two leader nodes. This completely fails the ACID specifically Consistency problems that RDBMS pride themselves in.

How does a Peer-to-Peer system such as CASSANDRA solve this issue:

In Apache Cassandra, there is no Leader. That means data that belongs to a node is asynchronously copied to other nodes to satisfy the Replication Factor set while creating Keyspace, which is generally 3. Each node contains full copies of data & in total the Cluster has 3 such copies.

While servicing the request, it is not necessary that the node which has the data will be servicing the request. Any other node can take that job, that node is called “COORDINATOR NODE”. It Coordinates with the node which has the Partition Key and acts as an “Exchange Point” between user and the server/node that holds the data.

Now consider the Cplit-Brain problem happening here and the Cluster has half of the nodes down. Remember the replication Factor is 3 and Cassandra will continue to service requests based on the “Consistency Level”. Two of the most used Consistency Levels are: 1 & Quorum: Meaning majority of the nodes in the replication factor i.e., 2 out of 3 nodes which hold data copies as shown in Replication Factor of 3 should acknowledge.

Hence even if the Cluster is Split in Half there are always 2 nodes which are needed to assure the Cluster is consistent. Once the nodes that are down are back online. As we discussed earlier, Cassandra used Hinted Handoffs & replays queries so that the nodes that are back online can come to the updated state of the remaining Cluster.

This mean there is no re-election to chose a Leader node, the split-brain problem is avoided. For example: Netflix might say Consistency Level = 1 for retrieving videos and Quorum for inserting. Since majority of the work is Reading with Netflix and Data inserted is low compared to read. We can create 3 logical data centers. Each for: Inserting, Reading and Analytics.

VIRTUAL NODES

In a perfect scenario for Cassandra we will never have to decommission or re-commission any node. But this is not a perfect world. Without Cassandra making any mistake there are several issues that might bring the node down such as: Hardware Problems, System Software Upgrade issues, Operating System fails to load properly, the system has been on for too long becomes too hot, Program Logic that can halt the entire node etc., Cassandra itself might not have been installed properly or configuration might have been done wrong.

Also for example while data is inserted few nodes which get latest data might be overloaded. This might happen if we do not use Partitioners like: Murmur 3 or MD5. Also when another node joins the Cluster, the Partitioner will stream data from overloaded node to the new node, this will put extra load on the node especially if it is the node servicing requests.

VIRTUAL Nodes solve this issue. Virtual Nodes when enabled, make each physical node as multiple virtual nodes. Earlier we have seen how two nodes have each split token range. This makes one node take the entire load of a request. Cassandra is capable of handling loads but these days as more and more people come online and more than half the world is still not connected to the internet. We already have a lot of demand and it is only growing. We have seen the demand shoot up during the pandemic for online deliveries. Imagine the amount of stress these nodes/.Clusters handle. If a single node contains responsibility for a CONTIGUOUS range of large hashes, it will get strained and even though in Cassandra each CORE can service about 12000 requests/second/core imagine a system with 24 cores. That’s 288,000 requests. Imagine Amazon, Netflix, Walmart, Microsoft, Apple which stream services such as; Delivery, Content, live stream etc., Every click is a request for new content and each user clicks more than few clicks at each website. Now multiple this by 10. That’s approximately 2.9 million requests. Now we can say cloud systems are big and capable. Yet they just don’t do only user servicing, memory and storage are spent for various reasons such as: System management, Converting user requests to system understandable format, loading videos, audios and images takes up more space than messages. Notice when we send GIF images or videos via iMessage there is BLUE LINE progressing to show how much the progress has been made to deliver your message. iMessage is hosted by Apple a secure service and handling more than 500 million handsets all sending videos, GIFs, text. Adding to this iTunes, iMovies, iCloud, iBooks, Games. Imagine the load a Cassandra system that supports these requests might be managing. It is said Apple has over 100,000 Cassandra nodes.

Upon all of this, Cassandra also enables OLAP using Big Data, Spark, Solr etc., Analytics from Apple such as Apple Stocks, trends Apple users to attract more customers not only helps users but also the system takes in more load from Apple itself to monitor behavior. Additional new services such as: Apple News only add to it. 😊 That is why Apple has 100,000 nodes. We read that each nodes can handle 2-4 Terabytes of Data. Multiply that by 100,000 that is 2.4\*10^6 bytes of data. Netflix on other hand hosts 2500 nodes Cluster with 420 TB of storage and serves 1 Trillion requests per day.

The stress that is on each node is tough to imagine with such systems. Hence Cassandra allows a single node to handle multiple “SMALL TOKEN RANGES” instead of “ONE CONTIGUOUS RANGE”.

A consistent Partitioner shares the load properly. Let’s assume 2 nodes are servicing the requests coming into a new community application. To manage stress when a new node is added i.e., Commodity HW with Cassandra, two nodes which has 2 virtual nodes each, now the data is divided into 3 virtual nodes and the two nodes stream data accordingly so that now each node contains 1/3rd of each other node. Since the new node is yet to start servicing. It takes in data much faster because it cannot start servicing until the load is completely shared. Based on the number of virtual nodes.

Default value for VNodes is 128. That means each node will contain 128 smaller token ranges.

VNodes automate the token assignment and takes the load off of the Cluster Administrator.

VNodes value can be changed in Cassandra.yaml file by changing num\_tokens value. Remember earlier when we set the token range for second node that is to signify which node will hold which contiguous token range. With this VNodes at 128 be default it meant that the 2 nodes were divided into 128 smaller nodes and data has been shared accordingly so that a single node doesn’t bear the brunt of requests. The highest number of VNodes for a powerful computer allowed is 512 num\_tokens, 128 is the lower bound and 256 is used in general by Medium & Large companies.

This means using VNodes is a new node is added while an existing node services the request. The existing node is not overloaded to stream a large chunk of data rather send in smaller amounts. Which will reduce stress on the memory and user will not expect any latency.

Earlier what we did was a STATIC assignments, MNC’s earlier used to add “2 NEW NODES” every times. So that load on each existing node is halved. Adding a single node does not make much difference hence earlier days we only used to Double the number of existing nodes. This helps scale the system and handle the nodes and increase servicing capability or scaling as we say.

The problem with STATIC SCALING is that once you agreed to keep doubling nodes. For a company like Apple, Amazon or Netflix. That meant once they reached let’s say 256 physical nodes/commodity system servers. The only other option is to bring in another 256 such systems. That will scale the system, but the existing 256 systems were accumulated over a period of time. Hence for the existing 256 systems to even be worth maintaining it will take probably another 5 to 10 years. That kind of defeats the argument as to why Cassandra prefers commodity hardware. Now to handle the pandemic and it’s after effects maybe it kind of makes sense. But in regular once things get to normal it does not make sense for a system designer or financially.

Virtual Nodes remove this need and by using PARTITIONER which can share the tokens/Virtual nodes evenly among physical probably we might then add maybe 10 new physical nodes rather than 256. Yet we keep the system consistent.

All the data that we have worked on so far or the Cassandra system stores is in the following directory:

/node/ubuntu/node1/data and similar for node2.

This directory contains 4 important folders:

1. Commit-log: Append-only data structure that adds to the speed because of sequential IO.
2. Data: Where the Keyspaces user defined and Cassandra System Keyspaces exist
3. Hints: As we discussed earlier, hinted hand offs are used to get the node updated and
4. Saved-caches: Builds and Shuts as the Cassandra Server runs and shuts.

Since we will be using VNodes going forward we have to remove all the data that existed till now.

This is done by deleting “data” folder. This does not mean the earlier configuration changes are also removed. They have to be changed once this folder is removed to set VNodes to 128 in num\_tokens.

Remember num\_tokens help set number of VNodes and initial\_token helps configure token range in a STATIC way.

As an additional NOTE, we configure hinted Hand Offs to YES in Cassandra.yaml file.

Steps to launch nodes and Cluster with VNodes enabled:

1. Stop node 1 and node 2 by using command /home/ubuntu/node1/bin/nodetool stopdaemon
2. Remove the data files in node 1 and node 2 using command: rm -rf data in folder /home/ubuntu/node1/ so for node 2.
3. Now open Cassandra.yaml file in the folder /home/ubuntu/node1/resources/Cassandra/conf, here set num\_tokens to 128 and comment off initial\_token. One is for VNodes and the second is for Static node configuration of token ranges.
4. Now restart node 1 and node 2 using /home/ubuntu/node1/bin/dse Cassandra, same for node 2. Observe node 2 takes more time to join the cluster and throughout you could see the Vnodes being created.
5. Now run the command /home/ubuntu/node1/bin/nodetool status, Notice Token Count is now 128.
6. Now run command /home/ubuntu/node1/bin/nodetool ring. This command details all tokens of a node.

GOSSIP

GOSSIP in Cassandra Cluster is a Broadcast Protocol. The purpose is to share information of the state of the node with up to 3 nodes in total. GOSSIP does not share data it shares STATE information of a node.

Each node does a GOSSIP message every second. There is no centralized server holding this information. Nodes/peers spread this information among themselves in the aim to maintain the latest state information automatically.

If a NODE is allowed to GOSSIP with any other NODE without control then because this is done every second and with the number of nodes ever increasing. In the case of Apple 100,000 node Cluster. The number of GOSSIP will be polynomial. Hence we limit GOSSIP to be with up to 3 randomly selected NODEs.

SEED NODEs have a higher chance of being selected for GOSSIP just to keep them in the know.

A NODE does not remember the NODEs with which it shared the State Information with. NODEs can come and go as they please and there is no single LEADER node that all others rely on.

GOSSIP allows an overarching data structure called “ENDPOINT STATE” to save the state information.

Endpoint State is an overarching data structure storing all meta-data about a node. This again has the following sub structures:

1. Heartbeat State: This data structure contains 2 values, Generation: 5 which contains timestamp of when the node bootstrapped and Version: Is a simple Integer value that increments every second. Heartbeat values increment every second and are shared across with other nodes via GOSSIP. This allows other nodes to decide if the nodes which it received the message from are UP or NOT.
2. Application State: This stored meta data of specific node that GOSSIP spreads throughout the Cluster. Some of the meta data that this data structure stores is:
3. STATUS: NORMAL?BOOSTRAPPING?LEAVING, whatever the status is of a NODE up or DOWN. Once the other NODEs receive the GOSSIP they get to know the status of the NODE but they do not GOSSIP this state to other NODES.
4. DC: WEST
5. RACK: rack1
6. SCHEMA: The Schema number changes as Schemas change over time
7. LOAD: Dis space usage etc.,

Each NODE stores meta-data about 2 other NODEs, it sends this GOSSIP to other NODE which through SYNC & ACK messages or HANDSHAKES sends the acknowledgement for the data it needs and the latest data it has. When each NODE shares latest information with another that message is called a DIGEST. Each NODE’s INITIATOR now compares which NODE data should be values and makes the update.

The advantage with GOSSIP is that, it keeps the Cluster stable with updated information and this sharing of information does not cause much network traffic.

SNITCH

A SNITCH in Cassandra reports a NODE’s topology i.e., The Rack and The Data Center. That means using SNITCHs we can determine which NODE belongs in which Data Center/Rack.

There are fundamentally 2 types of SNITCHs, the Regular Snitch and the Cloud Snitch.

In Regular SNITCHs the most popular is GossippingPropertyFileSnitch. But there are others such as:

1. Simple Snitch: It places all nodes in Data Center 1 and Rack 1. Rack and Data Center are Logical Assignments which you would usually want match to your Data Center.
2. Property File Snitch: Instead of having the Data Center & Rack hardcoded like Simple Snitch, Property File Snitch picks its configuration from a file called CassandraTopology.properties. All though this sounds simple imagine for Apple that has 100,000 Cassandra nodes. It is upon the administrator that this file exists on every node and all the data is in sync.
3. GossippingPropertyFileSnitch: In this Snitch you configure the rack and the data center for an individual node and the GOSSIP protocol takes care of sharing the data with NODEs of the Cluster. File name: Cassandra-rackdc.properties file and
4. RackInferringSnitch: This tried to infer the NODE’s Rack and Data Center from the IP address of the NODE. Unless the IP configuration is clean enough to match the physical setup of the Hardware it is not advisable to use this snitch.

Cloud Snitches are as follows:

1. Ec2Snitch: Single Region Amazon EC2 Deployment,
2. Ec2MultiRegionSnitch: Multi Region Amazon EC2 Deployment,
3. GoogleCloudSnitch: Multi Region Google Cloud Snitch and
4. Cloudstack Snitch: For Cloudstack environments.

Dynamic Snitch: This snitch comes with the system and acts as a wrapper on the snitch we configures in cassandra.yaml file. Its job is to monitor Cluster Health and Performance. When a NODE needs to interact with NODEs that carry copies of the data, the Dynamic Snitch makes sure to check which of the NODEs are in a better “state” to handle the request.

We must be sure all NODEs in the Cluster use the same Snitch. If not there with be inconsistency. If it is necessary to change the type of Snitch being used, we must restart each NODE in the Cluster. Once the Cluster is back up we should run Repair and Clean-up on all nodes.

Use Cassandra.yaml file to change the Snitch from Simple to GossippingPropertyFileSnitch, before that DOWN the nodes/cluster, delete Data in all nodes. Then change the Snitch type. Change the Cassandra-rackdc.properties to change Data Center and Rack Information.

REPLICATION

Apache Cassandra targets High Availability, which means it should make sure once the system design has been complete and data model is done. The system once brought on will eb rarely brought down.

Assuming we did not replicate even though Cassandra allows us to do so. If that node goes down, we will lose the data and like RDBMS will have to wait for fail over. This is not desirable in Cassandra.

Hence the general replication factor is 3. Every time user enters data that request and response management job falls to a COORDINATOR node, that node because it is aware of the topology and over time we save the details in the driver as well. The Coordinator Node makes sure that it replicates the data to nodes which own the Hash Ranges.

Since the replication is 3, the data is written by the Coordinator node to not only the NODE that owns that range but two more nodes after it that will own the range for redundancy.

Also now that we use VNodes set to 128 it will now be upon the Partitioner to determine the nodes to which the data will be written to.

In a single data center, this might be easy. But Cassandra is designed to span across Data Centers be it logical or physical right out of the box. Assuming we are creating 2 Logical Data Centers, that means we are splitting the Nodes/Hash Ranges into two logical groups. Which is a neat grouping.

Now Cassandra uses the Snitch to make sure data is replicated across data centers and also because through GOSSIP it is topology aware.

Assuming we are creating a Keyspace, we agreed to have a Replication Factor of 3. We set this Replication Factor while creating the Keyspace, where we specify which Data Centers will hold the Data and its Replicas as follows:

1. Create Keyspace Killrvideo with Replication = {‘Class’:’NetworkTopologyStrategy’, ‘dc-west’:3, ‘dc-east’:3}

Earlier when we created Keyspaces, we used SimpleStrategy and ReplicationFactor of 1. But now we have 2 Data centers. So when the user data is sent:

1. Coordinator node on Data Center 1 receives the Data.
2. It will write data to the Nodes in the current Data Center.
3. The coordinator in Data Center 1 then asynchronously writes data to a single NODE in another Data Center which then becomes the Coordinator for that Data Center and replicates the data.

Once you create a Keyspace and some tables with data. You can use them to query and see which partition exists on which node:

./nodetool getendpoints killrvideo videos\_by\_tag ‘cassandra’;

The thing here to note is partitions Cassandra and DataStax exist but to check which Node/IP does Cassandra or will Cassandra insert data in the future, you can do a test for that. Since the keyspace will be the same and data will change try using another Partition Value other than ‘Cassandra’ or ‘DataStax’.. It will show the nodes the data will be added to. For example:

./nodetool getendpoints killrvideo videos\_by\_tag ‘yourname’;

You will see the IP of nodes on which the data will be inserted to.

Remember in earlier scenario while changing data center information in Cassandra-rackdc.properties, we assigned it as a part of data center on which there is already node 1. Now the 2 data centers would be 2 clusters 1 with 2 nodes and another with 1 node.

As when creating keyspace we defined that data should be replicated in both ‘east-side’ and ‘west-side’ data centers. We will always see 2 copies of the data if we use a replication factor of 1, but generally people use 3.

CONSISTENCY

Consistency is a paradox if you ask me. Because when issues like Split brain problem occur. RDBMSs do fail-over and re-elect a leader node. Moreover if the data is sharded it will create synchronization problems keeping other dependent process waiting for fail-over and leader reelection to complete. In some cases it might take a minute to few minutes and in other cases it takes a day or two.

CAP theorem says when given a choice of Consistency, Availability and Partition Tolerance, where RDBMSs choose Consistency and Partition Tolerance. NoSQL system especially Cassandra chooses Availability and Partition Tolerance.

RDBMSs over come Partition Tolerance through fail-over, reelecting a Leader node.

Cassandra says Consistency is a Paradox. For example:

1. Request for updated data from client might reach the follower node before the Leader node complete synchronizing data along with handling requests coming in.
2. When there is a network split and Leader nodes go down, the Follower nodes re-elect a new leader. What would happen if the Leader node that went down came back online. Now the system will have 2 Leader nodes.
3. Sharding data will need special tools/scripts to be written to monitor where a certain Horizontal Partition is stored and if the script does not have the facility to remember it, it would have to search all shards. Even the search script itself is an overhead. This will only add to the Latency not reduce it.
4. As RDBMs scale denormalizing data or changing data model to scale might happen, which again begs the question why so much trouble in scaling?

Cassandra chooses Availability saying if the Node is highly available & even if the node goes down the system can continue servicing through other nodes that store data copies. It choses Partition Keys which help find the node holding the data in constant time. It also has append-only data structures such as commit-log adding to the speed with sequential-IO. It also does not do in-place add or delete, it creates a Tombstone for each deleted record with a timestamp. It uses compaction and read-repair to keep the most updated data. This takes advantage of Partition-Tolerance and Availability.

This does not mean it gives up Consistency. It allows users to choose how much consistency they want the system to be. For example: If the application developer is a bank. The design could be of 2 data centers, within which there are 2 logical data centers. One for OLTP and another for OLAP. This way if One data center goes down another can pick up servicing right away. Because the IO is so fast it also would not show much to any Latency when node goes down.

Cassandra allows Financial Systems to choose highest consistency but also allows them to run analytics with consistency level 1. Both while the system/Cluster is highly available, with highly consistent logical data center One used to confirm transactions and the logical data center two is used for analytics in near real time. Remember this is on 1 physical data center. Even if this whole Data center goes down, the 2nd data centers continues the service without any interruptions and while the team works to bring back the data center that went down back up.

Consistency levels are part of a query not while creating a Keyspace. Keyspace mentions Replication Factor and Read/Write queries mention “Consistency Levels”.

With Replication Factor at 3, the minimum Consistency Level of 1 can be used in OLAP platforms for super fast analytics and consistency level all can be used for OLTP.

But in general on Cassandra the most famous Consistency levels are: One and Quorum. Meaning majority of the nodes should acknowledge that data has been received or acknowledge request for data while the node responsible for retrieval sends data back.

Consistency Level of ALL is not advisable unless it is an absolute case. The problem would then be it would be similar to RDBMS and the system can no longer service requests even though the data is available on another active node.

Hence to keep with both Availability & Consistency it is advisable to use QUORUM as recommended Consistency Level. Then we call system to be Strongly Consistent.

For a system like Netflix they could use Consistency Level Quorum to make sure the Data Writes have been successful and to read a video they could use Consistency Level 1. Because we have acknowledged that data has been successfully reading back the same with Consistency Level of 1 is not much of a problem. Also Cassandra is eventually consistent, meaning if it is used in a time series data like traffic jams at certain time, even if the data comes in a second or two later. It does not make much effect or will have no effect. Hence Cassandra system adds to the speed and consistency and allows the architect to tune it to their comfort.

These are all good for a single data center but what about the consistency level of data across multiple data centers?

When we configure a Keyspace, we mention which data centers should hold the data and what the Replication Factor should be. So when the data is written the coordinator node has the responsibility of writing not only to the node which holds the hash value but also to the node in the other data centers. Lets say we have 2 or 3 data centers. Now when the data is being written, assume by Apple users in form of Podcasts, comments, video and audio uploads etc., and assume we use a consistency of Quorum for the overall query. That means we are telling the system to wait until not only the current data center confirms, but also for all other data centers to confirm. This causes wait/latency. To manage this another type of consistency level called: LOCAL\_QUORUM exists. This when used only waits for the current data center/Cluster to confirm and the Coordinator node continues to write the data to the other data centers “Synchronously”. This saves multiple microseconds, which in terms of services like iTunes, Netflix or iMovie is a huge advantage/addition to scalability.

The consistency levels that can be used across data centers are:

1. ANY – A hint that data is stored is satisfactory,
2. ONE, TWO, THREE – Check closest nodes to the coordinator,
3. QUORUM – Majority Vote = ( (Sum of Replication Factors/2)+1),
4. LOCAL\_ONE – Closest node to the Coordinator node in the current data center,
5. LOCAL\_QUORUM – Closest Quorum of nodes in the current data center,
6. EACH\_QUORUM – Quorum of nodes in each data center, applies to writes queries only and
7. ALL - Every node must participate.

HINTED HANDOFF

HINTED HANDOFFs are a special sauce of the Cassandra system. When a Cassandra node goes down to which DATA had to be written. For example: When Replication Factor is 3 and one of the NODE is down. The Coordinator Node saves “Hints”. Then when the DOWN Node comes back up the Coordinator Node writes back the data to the Node. The Hints are by default stored in the Coordinator Node for 3 hours and can be tweaked.

The Coordinator itself could go down and the NODE that needs data be written might come back online. Using Consistency Level ANY is not advisable. Because then the Coordinator without acknowledging from any node replies that the transaction is successful.

If there is a delay of more than for 3 hours either because the NODE went down or the Coordinator went down before the node could come online. A Read-Repair or Cleanup need to be done.

READ REPAIR

In Cassandra we choose Availability over Consistency. That means when there is a Network Partition we can continue to serve requests without having to wait for fail-over and have down time.

Cassandra also has Tunable consistency, that means when the System has a Consistency Level of all, the Coordinator Node which handles Request and Response on Cassandra, has to wait for all nodes to respond. While the Node with least activity is chosen to service the request, there are chances of the system being in-consistent overtime in Cassandra. Hence when we use Consistency level of All lets say in a Banking Application, the coordinator Node receives data from one Node and a Checksum from other two nodes. If the Checksum matches, the data is returned to the user. If it does not, the Coordinator node compares and sends the Checksum with latest timestamp data back to the user, it then deletes the other two Checksums and sends the most recent Checksum to both the nodes and updates them.

This obviously causes issues of speed & latency in Cassandra. Though we might choose a system where we can set Consistency Level to All, it is generally not advisable and depending on system to system Banking to Content system for example, they would need different Consistency Levels, for example a Banking Application can use Quorum to both Read & Write, whereas Netflix can use Quorum in write & One in read.

Even in cases where there are inconsistencies that might prop up Cassandra allows us to do a Read Repair approximately 10% of the time (settings can be changed to have higher % of read repair). When a request arrives Cassandra retrieves data, sends it back and does Read Repair asynchronously 10% of the time. Most of time the data is latest but there might be old data in cases but immediately after the old data is sent. The Cassandra system makes the read repair bring the system to latest data.

When needed we can use tool such as: nodetool repair, but this might put a lot of pressure on the system based on the data model you created which might have led to many read repairs. This should be used as the last line of defense.

Nodes that are not read that often also might benefit from a read refresh from time to time.

NODE SYNC

We can perform full Cluster repair via “nodetool” this often puts a lot of load on the Cluster and might cause problems is servicing requests. Depending on how out of sync the Cluster nodes are this might put additional load on the network as well. It is also upon the Administrator to perform a Full Repair which might include all nodes within gc\_grace\_seconds to prevent Zombie data.

DataStax tool called NodeSync which is a better version continues to run in the background which puts minimal load like GOSSIP protocol and brings nodes with out of data to the most update information.

This feature is only available on the DataStax Enterprise and must be enabled on a per table basis. All nodes run Node Sync in the background, checking on their replicas to ensure they are in SYNC.

Syntax: Create Table Table\_Name (….) with nodesync = {‘enabled’:’true’};

NODE SYNC divides token ranges into segments, it monitors and repairs them as a group. It also tracks its success or failure saving its progress as it moves along. Each time NODE SYNC completes the repair on a SEGMENT. It records its progress in a Data structure called “Savepoint”.

GC\_GRACE\_SECONDS is the time before which the administrator has to perform a full repair. By default it is 10 days. NODE SYNC prioritizes SEGMENT repairs so that each SEGMENT is up to date before this time period. Taking the worry of repairing the entire cluster away from the administrator.

The default SEGMENT SIZE that NODE SYNC repairs is about 200 MB. If a partition is bigger than 200 MB then the Segment Size grows to the full size of the partition. To determine which Hashes become a single segment Node Sync determines that by considering the entire token range a segment and recursively splits it until the target size is found.

NODE SYNC assumes a decent data model with good distribution of data. NODE SYNC considers system repair as an Atomic Unit. The entire segment must be successfully repaired in order to consider the repair successfully completed.

NODE SYNC saves the status of each repair in the NODE SYNC status table. If repair of a segment fails during the process. NODE SYNC considers the repair un successful. The table in which NODE SYNC records if the repair has been successful or not is nodesync\_status.

Some of the possible results of a Segment repair:

1. Full\_in\_sync: All replicas in Sync,
2. Full repaired: Some repair necessary,
3. Partial\_in\_sync: Not all replicas responded (at least 2 did), but all respondents are in Sync,
4. Partial repaired: Not all replica responded (at least 2 did), with some repair needed,
5. Uncompleted: One node responded/available & no validation occurred and
6. Failed: unexpected error happened check logs.

NODE SYNC uses the same process read repair or full repair uses:

1. Read data from all replicas,
2. Check for sync and if not update
3. Check nodes with latest timestamp data and update remaining nodes.

WRITE PATH

Cassandra is called a WRITE INTENSIVE work load manager. One of the reasons it is so fast is the WRITE PATH. In most DB systems the way they data is written is proprietary but with Cassandra it is an open Book.

When data comes in to write, Cassandra writes this in 2 places: MemTable and Commit Log. The Key difference is that MemTable is ordered by Partition Key followed by Clustering Column. Whereas Commit Log stores data sequentially, every data appends to the end of the log each time. At this point Cassandra acknowledges to the user that the Write is successful.

Since MemTable is always in memory and we store data as Partition Key followed by Clustering Columns. Cassandra can continue to read & write from memory and we all know that reading from memory is order of magnitudes faster than Disk.

The purpose of the Commit Log is if the Node goes down for some reason. We can still have the data. Once the node is back on, in order to get the node back into the most updated fashion. Cassandra reads Commit Log off the disk & replay all the mutations to get the Node to current level.

Hence MemTable is to READ data and Commit Log is to restore the node.

We all also know that Memory unlike Storage is Finite, Cassandra allows the administrator to configure how much data should be in-memory but eventually it will get Full. Apache Cassandra then needs to get that MemTable off of Memory and on to Hard Disk.

At this point both Commit Log and Data written from MemTable both exist on Disk. Hence at this point we do not need the Commit Log since Cassandra wrote it back to the Disk. The new Data Structure that is created when MemTable is written to Disk is called SSTable i.e., Sorted String Table it literally means that data is partitioned based on Partition Key and is Followed by Clustering Column.

An SS Table like the Commit Log is IMMUTABLE. The recommended method is to use separate disks for Commit Log and SSTable. The reason is while Read and Writes continue to happen & SSTables are being created all the time. Having a separate disk for Commit Log which simple does an append each time will reduce the load on the disk that has the SSTable.

READ PATH

When we move data from MemTable to SSTable every time the memory is full, we create multiple SSTables. This also means that the data is dispersed among multiple SSTables and also in the current MemTable. This means Cassandra needs to coalesce all the data to present it to the user.

While reading data remember we use Partition keys, which means data in MemTable and SSTable is grouped together based on Partition Key & the Clustering Columns. Also the memory is super fats the disk. When the request comes for Data, a Binary Search is performed for the Partition Key value and the data is returned (after some filtering of course for Clustering Columns and Where Clauses).

When it comes to an SSTable it is a bit different:

When data is grouped by Partition, certain partitions might be bigger than the others. Hence when they are put on the SSTables each table might of different size. When stored on disk they have “File Offsets”. Partitions are stored in the File and Partitions all do not have the same length. Starting from byte-offset 0 to end of file 1 and so on.

On the SSTables there are few other files apart from the data that help retrieve the data, they are:

1. Partition Index Table (Stored on Disk): It literally has a Key-value pairs of Partition Key value to the byte offset of the file at which the data resides. Seek to that location on the SSTable file and stream the value back to be Coalesced with other values retrieved from other SSTable files. We also remember that partitions can be one too many and can get quite large.
2. Partition Summary (in-memory): Built on top of partition index. It is also a Key-Pair table but instead it has as the key “Token-Range” & the value would be byte-offset. Byte-offset is an index. So when a request comes in Partition Summary shares the byte-offset index, the seek is done to that index & byte offset. Moves comparing one by one until the record is found of the actual byte offset on SSTable file and the data is sent back to be Coalesced with other data,
3. Key Cache (in-memory): Key Cache is the record of most recently seen SSTable File Byte offsets, stored as Key value pairs in-memory. When a request comes in Cassandra checks this in-memory data structure if the File offset can be found. Then it makes a direct seek to that location in the SSTable File byte offset and streams back data to be coalesced .
4. Bloom Filter: A Bloom filter tells us if an element is definitely not in the Set or that may be it might. Which is when we continue to search for it. Some times it is possible to do False Reads which is when the Bloom Filter does not know if the data exists. Which means we have to SPEND a read without a result. If we can afford a bit more space in-memory we can get better results using Bloom Filters.

All this is available in open source Apache Cassandra. DataStax Cassandra comes with much more optimizations. In the DataStax Cassandra:

1. We do not have the Partition Summary data structure.
2. Partition Indexes are changed to TRIE based data structures.

Using DataStax will make read look-ups super fast because of this. Migration from open-source Cassandra to DataStax Cassandra is seamless, old SSTables are compacted and brought into the most updated version and removes Partition Summary and makes Partition Index a TRIE based data structure.

COMPACTION

In a NODE a lot of data comes in and gets deleted. This creates multiple SSTables and with records that are Stale and are no longer needed. An SSTable is immutable that means it is also un-editable, that is why we say there are in-place edit or delete in Cassandra. When data is deleted, a special record called Tombstone is created. Which means the record for that data has been removed.

When Compaction occurs between 2 SSTables records of each matching Partition is compared & the record with the latest timestamp is kept. For example if the data has been deleted recently, every tombstone has a Grace period of 10 days. Any Tombstone data more than 10 days is automatically deleted from Cassandra while performing Compaction. If there is a Tombstone and the grace period had not expired the tombstone is written to the new SSTable created by comparing the records is 2 SSTables. If there are records which are not updated for quite long they are written to the SSTable as it is.

The reason we keep the tombstone of the grace period is not expired is, because if a read-repair needs to be done. The Tombstone will reflect that the data has been deleted as of latest records.

Once the new SSTable has been created we delete the 2 old SSTables as they are no longer needed.

This is what happens with the partitions when they are merged to a single SSTable. But an SSTable can contain multiple partitions and each partition can be of a different size. Hence each SSTable with its corresponding partition is compared, stale data is removed if it goes over grace period time of 10 days, if not it is kept as a tombstone or an actual record.

There might just be a single partition overtime, hence when compaction happens the single partition is checked to see if the data is over grace period with a tombstone if not the data is kept.

Now there can many SSTables which need to be compacted putting a lot of load on the performance. To help with that Cassandra has 3 compaction strategies, to choose which SSTables need to be Compacted. They are:

1. Size- Tiered Compaction: This is the default type. It takes SSTables of the same Size and does Compaction. This strategy is great for a write heavy workload,
2. Leveled Compaction: Good for READ heavy workloads. Groups SSTables into Size-Levels. Where each level is 10 times bigger than the previous Size-Level and
3. TimeWindow Compaction: If data comes in sequentially by time. Creates time windowed buckets of SSTables which are then compacted using Size-Tiered Compaction.

The setting is to be done at Table level. Syntax:

ALTER TABLE TABLE\_NAME WITH COMPACTION = {‘class’: ‘LeveledCompactionStrategy’};

ADVANCED PERFORMANCE IN DATASTAX ENTERPRISE

DataStax 6 and on have huge performance gains by introducing a basic change in architecture. This is based on how DataStax uses Cores within a Single machine. Generally to scale DataStax enterprise we add more nodes.

In version 6 and on, DataStax takes advantage of Vertical scaling by assigning a thread to each core and assigning the jobs to that thread. On the other hand open-source Apache Cassandra uses Thread Pools. It creates a Thread for each request that comes in and replenishes that thread once the task is complete and adds it back to the Thread Pool. Here there is a problem of Thread Contention.

Thread Contention meaning as new tasks come in Threads are assigned from the pool. The OS thread management system kicks in as a result. This allows the OS to be in control and the OS can change the Affinity of a Thread-Core as the request is being processed from one core to another. This needs the new core to read the data required to process request again.

These days we have multiple cores per CPU and the more threads there are, the more management and maintenance overhead there is.

DSE only creates 1 thread per core instead of having a pool available to manage. Hence the thread maintains the affinity to the core and takes in new processing request as it continues to complete each task. This also reduces the overhead of management and maintenance. These threads also never block.

When it comes to WRITES and READS both are asynchronous. In DataStax Enterprise 6 and on, each Thread gets its own section of the MemTable to write to, removing chances of locking which can happen when resources are shared. This also make all accesses Serialized which is another huge performance gain.

DSE 6 leaves 1 core for low contention tasks like: Compaction, Flushing MemTable to disk, Hints and Streaming.