International Journal of Management, IT & Engineering
Vol. 9 Issue 3, March 2019,
ISSN: 2249-0558 Impact Factor: 7.119
Journal Homepage: http://www.ijmra.us, Email: editorijmie@gmail.com
Double-Blind Peer Reviewed Refereed Open Access International Journal - Included in the International Serial Directories Indexed & Listed at: Ulrich's Periodicals Directory ©, U.S.A., Open J-Gage as well as in Cabell's Directories of Publishing Opportunities, U.S.A

EXAMININGCASSANDRA CONSTRAINTS: PRAGMATIC EYES

Sameer Shukla*

Abstract

Modern world applications are expected to be highly available and responsive plus the data volume is huge, traditional databases have the problem of scalability. Cassandra is one such database which is highly available because of its distributed nature and it is scalable we can add or remove nodes easily from the Cassandra Cluster. But it comes with certain restrictions and each fundamental of Cassandra (reading, writing, partitioning) has the answer to common questions like why Joins, Like Queries, Aggregate Functions are either not allowed or not recommended in Cassandra, to incorporate Cassandra in the application we need to understand them clearly as it affects engineers big time coming from RDBMS world. This write-up focuses on understanding those constraints and using them to design better database models and queries in Cassandra.

Keywords:
Cassandra;
Distributed Systems;
Reactive Systems

* **Master of Computer Applications,Bangalore University,6201 Breeze Bay Pt, #1634, Fort Worth, Texas-USA.**

1. Introduction

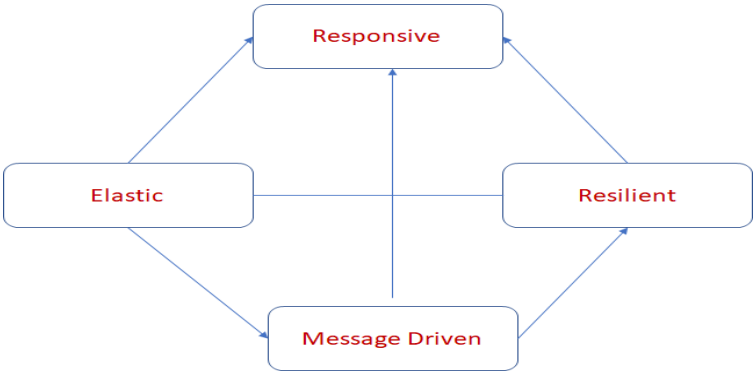
Cassandra is a distributed and decentralized database; it is scalable because it is distributed. Some of the key features of Cassandra

- No Single Point of Failure: Cassandra is a fantastic database, it has no single point of failure it is decentralized, that is if one node is down in a cluster there is no effect on the client accessing it as the request will be redirected to other nodes to serve.

- Scalability and Elasticity: Discussed below in detail as it's the most important reason why I selected Cassandra.
- Fast Writes: Writes are free in Cassandra, it performs fast writes and can store terabytes/petabytes of data without sacrificing the read efficiency, provided it is modeled properly.
- Visualize Cassandra as a Sorted Map Data Structure. The Key of the map is the row-key and value of the key is again a sorted map. Map<RowKey, SortedMap<ColumnKey, ColumnValue>>

2) Cassandra is Scalable and Elastic

The modern-day applications comparatively becoming much more complex, that's because the requirements have been changed a lot as compared to the last few years. Server nodes are scaled from tens to hundreds, average response time expectations now are in a second, the much better user experience is expected, data volume is in Terabytes. To develop such high availability systems companies are adopting reactive programming model and reactive manifesto has four important core principles and one of them is Elastic.



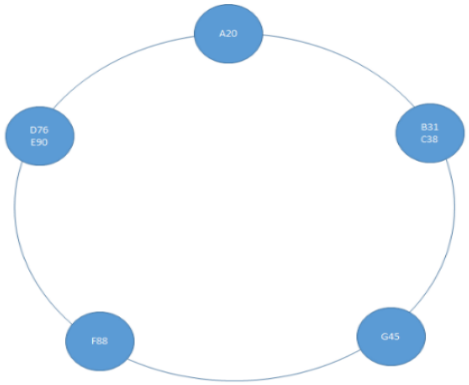
Cassandra Architecture is elastic because it is distributed and decentralized, we can add or remove nodes from the cluster quite easily if the volume goes up or down and since it is distributed in nature it is highly available if one node goes down, the query will be served by other nodes in the cluster.

3) Anatomy of Cassandra

Partitioning Key

For now, consider Partitioning key is a Primary key although Primary Key in Cassandra is (Partitioning Key + Clustering Key), Primary key uniquely identifies a row in the column family.

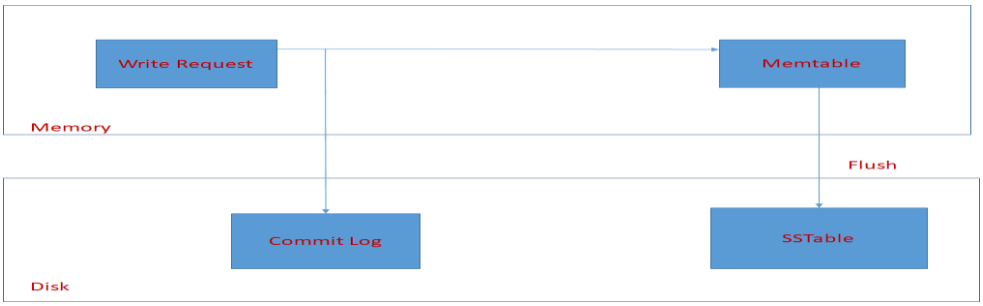
The purpose of Partition Key in Cassandra is a little bit more than uniquely identifying a row, it also identifies the partition or node in a cluster that stores the row. When data is written in Cassandra a Partitioner generates a hash value (token) this hash value determines the node where the row is going to be stored during writes and this hash value is again computed to identify the node/partition during the reads to locate the row. Partitioning key can be composite. Let's model Flight Management System in Cassandra where right now consider Flight Number as our Partition Key



The above image shows a 5 nodes cluster and the data with Flight Number as the Partitioning key is distributed across each node.

Handling Writes

When the user submits a write request, Cassandra stores the data in memTable and Commit Log simultaneously, memTable is a cache and once it is full, Cassandra invokes flush and the data is written from the memTable cache to SSTable (Sorted Strings Table) in sorted order of row keys in a disk. At the same time commit log cleans out all its data after a node writes the data notification is sent to the coordinator node about a successful write operation



Constraint: Avoid Frequent Updates

Its Last-Write win policy in Cassandra because of the way data is written it is important to avoid updates in Cassandra if there are too many as it impacts read performance directly. Data is scattered in too many SSTables eventually in each node and for reading the data the with the key which is being updated many times, Cassandra needs to read all those files and find the record with the latest timestamp to return the data. More details in the Modeling Section.

Constraint: Avoid 'IN' Clause

Over the period data will be added data to Cassandra and partition grows from A to Z and we have Terra bytes of Data in our system and we decided to perform IN query as

select * from flight_mgmt where initial in (A, B, C, D, E……Z)

In this scenario the coordinator node while serving the response to the user will keep all the responses in the heap because Cassandra may need to read many SSTables across the nodes and if one of the queries fails or timeout's in giving the response, Cassandra will retry it and again it's a memory intensive operation as Cassandra needs to read the entire partition data. It is the same scenario what we have discussed earlier, we should not try to use such queries in an RDBMS way the best practice here would be to query each partition separately.

Cassandra needs to pull everything if we use IN/aggregate functions such queries are timeout candidates

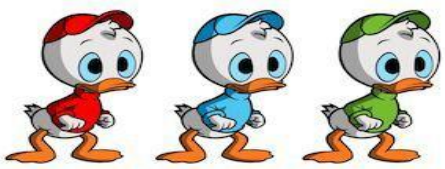
Constraint: Avoid aggregate functions

Take an example of count (*) aggregate function, in Cassandra it's a very expensive operation by count (*) means count all the rows across each partition/node. Each node has multiple SSTables imagining we have data in TB's so count query technically needs to read all the SSTables and keep the data in memory. 90% of the time this query will timeout because Cassandra throws Timeout exception if the result is not returned in 10 seconds. For a small cluster it's not a problem you might get a result but there is no guarantee that count is valid, while a count query execution is going on and a delete query has been performed by other user but delete is also an insert, tombstones is inserted into the table resulting in wrong count result, similarly while count query is going on multiple users may inserts new records or create new partitions in the database again resulting in wrong result.Same stands valid for other aggregate functions like SUM, AVG, GROUP BY (Invalid in Cassandra). Avoid them always as they are pure Timeout candidates. Keep the above image in mind always while using count or any other aggregate function

Partitioner

The partitioner is the hero who determines how to distribute the data in the cluster. A Partitioner algorithm converts the data's primary key (flight number in our case) into certain hash value say 21,31,155 and then looks at the token ring and allocate a node. And if a replication factor is of 3, then the same data is copied to two other nodes as well

Constraint: No Like Queries in Cassandra



Like is not allowed in Cassandra, the reason is simple: we cannot query Cassandra without partitioning key and partitioning key is a token a hash value so how can we perform Like on Unordered Tokens (Hash Value) 😊. Explaining below two important types of Partitioners a) Murmur3 and b) ByteOrderedPartitioner

Murmur3Partitioner (Multiply and Rotate)

Class: org.apache.cassandra.dht. Murmur3Partitioner is the Default Partitioner.

Murmur3Partitioner distribute tokens in an unordered manner. Murmur3Partitioner ensures that data is uniformly distributed across the cluster based on Murmur Hash Values. It's the most accurate one in distributing the tokens as compared to the other Partitioners, the Main reason for Cassandra achieving Partition Tolerance is because of this Partitioner.

Constraint: No Range Queries on Partition Key

The downside of the Partitioning mechanism is, one should avoid range queries based on partitioning key because tokens are distributed in an unordered manner.

For example: if your Partitioning Key is date and you want to display all the flights before 2019-01-02.

select * from flight_mgmt where token(arrival_time) > token ('2019-01-02').

Avoid doing such queries using Partitioning key as you will not get correct result-set as the reason I have explained above. But you can perform range queries on Clustering key, more details later in the modeling section of the post.

ByteOrderedPartitioner

ByteOrderedPartitioner is for ordered partitioning.

select * from flight_mgmt where token(arrival_time) > token ('2019-01-02').

This query now will work fine and will get the correct range results.

Constraint: Improper Partition Balancing and Hot Spots

This Sequencing comes with the cost of Improper Partition Balancing, Hot Spots that's why Murmur3Partitioner is the default Partitioner and should always be used.

Replication Factor and CAP Theorem

Let's understand Replication Factor from the query standpoint if you refer the Cluster diagram above flight number C85 is placed in Node 2 now what if this node goes down and the user tries to query

select * from flight_mgmt where number = 'C85'

The user will still get a response because that's where the Replication Factor comes into the picture. If in your system, the replication factor is 3 that means data (C85) will be copied to 2 other nodes and it will be replicated in a clockwise fashion to neighbor nodes.

CAP Theorem

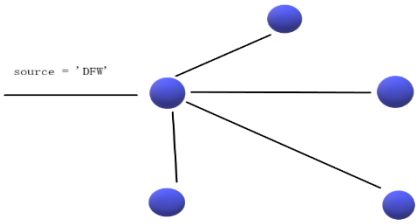
CAP stands for Consistency, Availability and Partition tolerance. In Distributed System Availability and Partition Tolerance is more important than Consistency. As expectation from Highly Available system is to be operational always, every request should be served regardless of the state in the requested node and that's what is expected from Reactive Application. Cassandra can be classified as an AP System (Availability / Partition Tolerance) but Cassandra can be tuned to achieve Consistency through Replication Factor. But we cannot tune Cassandra into a completely CA system because partition tolerance cannot be sacrificed.

Partitioning Key Restrictions

Assuming a table name 'flight_mgmt' with columns

create table test. flight_mgmt (number text, source text, destination text, arrival_time timestamp, departure_time timestamp, PRIMARY KEY (number))

- Querying Non-Partitioned Columns



If we want to query this table where we want to find all the destination where the source is 'DFW', meaning return all the destinations of the flight originating from Dallas Fort Worth.

select destination from flight. flight_mgmt where source = 'DFW'

This query will fail with the following 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"

Obviously, this query will fail because we have Flight Number as our partitioning key and data will be looked up when we use partitioning key in the where clause. Assuming Cassandra allows such query in that case if a request goes to node 2 then that node will try to check the source in all the other nodes and if anyone node goes down then the entire request will result in an error. Although it seems little difficult to understand this at first because we have only 5 nodes in the cluster but as mentioned earlier that if the data grew up to Petabytes and our node increased from 5 to 500 then this restriction imposed by Cassandra makes total sense.

Since there is a high possibility that such queries will result in a timeout, that's why Cassandra ensures that only Partitioning Key + Clustering Key can be the candidates of Where clause in the query, no other columns are allowed.

- Not Querying with all Partitioned Columns

Another Restriction with Partition key is, say instead of a number as PK we have two Partitioning Key's (number, arrival_time) and if we use the only number in where clause, Cassandra will not allow us to do. The reason is simple because Murmur3Partitioner will hash out this combinational value to identify the node and partition, so for locating the row also we need to provide these two columns in where clause as well.

select destination from flight. flight_mgmt where number = 'A20'

"Partition key parts: number must be restricted as other parts are"

select destination from flight. flight_mgmt where number = 'A20' and arrival_time=???

work's fine

Even while using 'IN' query just like the above scenario both flight number and arrival times are required else the same error will come what we have seen above
"Partition key parts: arrival_time must be restricted as other parts are"

- Using Order By

select destination from flight. flight_mgmt where number = 'A20' and arrival_time=??? Order by arrival_time

"Invalid Request: Order by is currently supported only on the clustered columns of the Primary key"

Ordering in Cassandra is only achievable through Clustering key.

Clustering Key

Clustering columns orders data (Ascending or Descending) within a partition. Consider a table with arrival_time as clustering now.

create table test. flight_mgmt (number text, source text, destination text, arrival_time timestamp, departure_time timestamp, PRIMARY KEY (number)) WITH CLUSTERING ORDER BY (arrival_time desc);

Data inserted in the table will be stored in sorted by descending arrival time

Clustering Key Restrictions

- In the Update query, clustering key columns are a must. You cannot perform update query without clustering columns also all the clustering key columns are required in the update query. If we try to perform update query without all clustering keys, the below error will be observed

"Some clustering keys are missing"

- In case if we have 2 Clustering keys, then while performing the range operation first clustering key is mandatory and second is optional. I try to perform range query with second clustering only, below error will be observed

“Primary Key 'second column' cannot be restricted as preceding column 'first column' is not restricted”

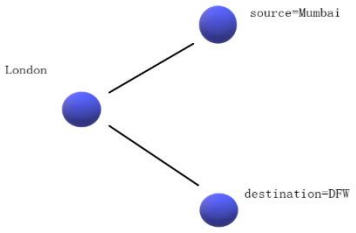
- Similarly, while using IN query with clustering key’s you cannot specify second clustering key column only, second is optional but first is always mandatory.

4) DB Modeling

Since we have covered some of the important basics and internals of Cassandra let’s start modeling our tables according to queries application needs to support.We are going to model a flight management system, let’s investigate some of the scenario’s

Scenario 1: Locate the current location of a flight ‘A20’, meaning the user will request to check what is the current location of any flight. Flight Started from Mumbai going to DFW via London.

select * from flight_mgmt where number = ‘A20’



The current model mentioned earlier where the number is our Partition Key will solve this query. But This model is an Upsert candidate as the only Number is the Key here no other Partition Key or Clustering key is defined.

Cassandra treats inserts/updates as same if the requested Partitioning key exists Cassandra updates else inserts. Initially, the table has only 1 row indicating that the flight is in Mumbai

insert into flight_mgmt (number, source, destination, arrival_time, departure_time) values ('A20','Mumbai','Mumbai','2019-02-04 00:00:00', '2019-02-04 02:10:00');

Once the flight reaches London, this row will be overwritten as now source is London and Destination is DFW.
insert into test.flight_mgmt (number, source, destination, arrival_time, departure_time) values ('A20','London','DFW','2019-02-04 10:10:00', '2019-02-04 14:20:00');

Point to remember, Upsert can be costly for reads, since Cassandra will need to look through lots of data on a single key and check whichever the newest one is, because the same row can span over a dozen of SSTables and for reads Cassandra needs to investigate all SSTables and find the latest one. Updates should be avoided if there are too many of them an application needs to support.

Scenario 2: Return all the historical data for any flight, so when a user queries the table with flight number ‘A20’ it should show all the locations i.e. Mumbai, London, DFW.

To tackle this situation, we need to tweak our model as it is not correctly designed to handle this scenario. There are two approaches we can follow, first is to break our partition key into two meaning apart from flight number add another column as a part of partition key may be arrival_time. In this way, we need to provide both flight number and arrival time as a part of where clause, but here the requirement in front of us is we will only be given flight number so our approach one failed here. Approach two just to handle this scenario would be to have a clustering key column adding clustering column will have a new row inserted every time rather than Upsert, also currently our data in the table is not following any order that is another problem we need to solve because flight started from Mumbai – London – DFW right now there is no order as well and making arrival_time as clustering key ensures the ordering of the data within the Partition ‘A20’.

insert into test.flight_mgmt (number, source, destination, arrival_time,departure_time) values ('A20','Mumbai','London','2019-02-04 00:00:00', '2019-02-04 02:20:00');

insert into test.flight_mgmt (number, source, destination, arrival_time,departure_time) values ('A20','London','DFW','2019-02-05 14:20:00', '2019-02-05 00:00:00');

Scenario 3: Return the History data of flight ‘A20’ greater than the given arrival time.

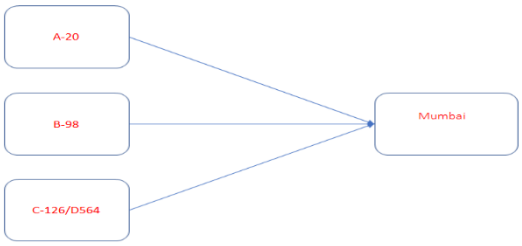
Range queries should be performed on Clustering keys, making arrival time as clustering key and then performing range operation on that will yield us correct results.

select * from test. flight_mgmt where number _number = 'A20' and
arrival_ time > '2019-02-04 00:00:00'

Managing Relationships (OneToMany, ManyToMany)

Scenario 4: For any given source, return all the flight details

In this scenario, the user will enter a source city name, like give me all the flights which are in Mumbai right now or in DFW right now. This is the classical example of OneToMany Relationship



We cannot satisfy this requirement with the current model as the 'number' is our Partition Key, not the source or destination. To support this query, we need to create another table with source as partitioning key and duplicate the records there as well or another option would be to have materialized view with 'source' as the partition key. I am calling the new table as 'flights_source'

```
CREATE TABLE test.flights_source (
        source text,
        number text,
        destination text,
        arrival_time timestamp,
        departure_time timestamp,
        PRIMARY KEY (source, arrival_time, departure_time)
) WITH CLUSTERING ORDER BY (arrival_time DESC, departure_time DESC);
```

Since both the tables ('flight_mgmt', 'flights_source') have the same set of data, the below query will return all the flights at a given source.

```
select * from test. flights_source where source = 'Mumbai'
```

Scenario 5: Return all the intermediate cities of flights A-20 and E-910 also return the entire journey of these two flights. We know flight A-20 is traveling from Mumbai to DFW assuming E-910 is traveling from Delhi to New York (ManyToMany)

To support the above query, we still need two tables. One our flight_mgmt table and create another table called 'flights_intermediate' and then we need to fire two queries separately as we know joins are not supported in Cassandra.

```
CREATE TABLE test. flights_intermediate (
        intermediate text,
        number text,
        destination text,
        arrival_time timestamp,
        departure_time timestamp,
        PRIMARY KEY (intermediate, arrival_time, departure_time)
) WITH CLUSTERING ORDER BY (arrival_time DESC, departure_time DESC);
```

4. Conclusion

Cassandra is a fantastic database with certain restrictions as it is designed in this way. In the age of gigabytes, petabytes of data reactive application development is the key and for developing that each and every layer of the application should reactit is only possible if we have distributed layers in our application, no single point of failures and elastic in nature. If we can tackle Cassandra's constraint efficiently as we have seen in the paper, our application will be performant and highly available.

References

[1]*https://www.reactivemanifesto.org/*

[2]*https://docs.datastax.com/en/landing_page/doc/landing_page/current.html*