Partitioning1:

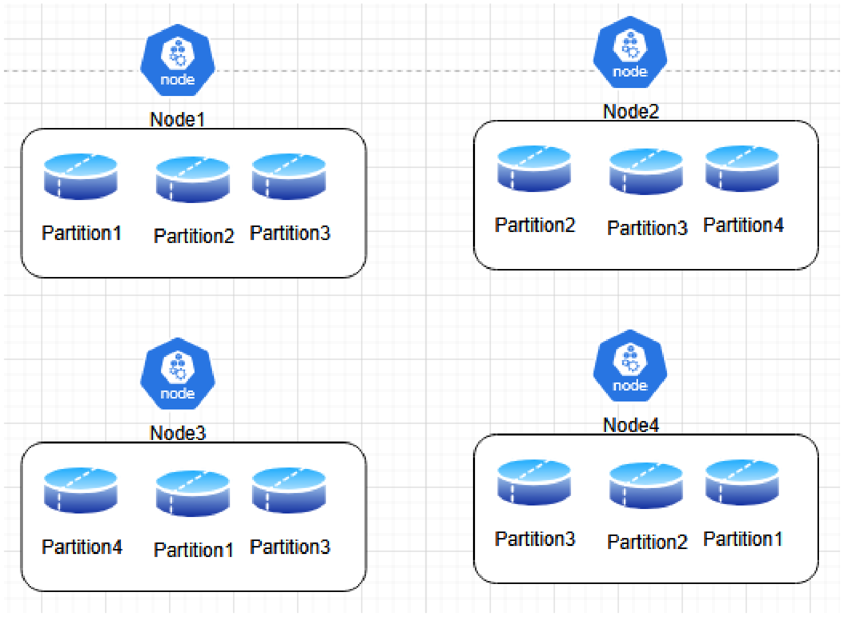
Partitioning is dividing our data into multiple nodes.

Why do we need Partitioning?
To scale our application we need partitioning

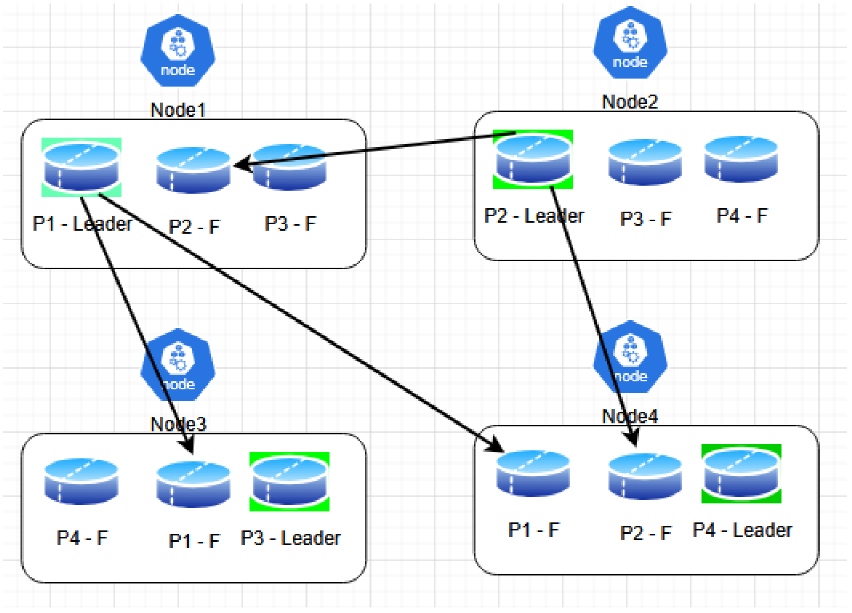
Replication is entirely copying the data and storing into other nodes. If data becomes too large, we cant put all the data into multiple nodes, so we instead the cut the same data into multiple parts and store them in different nodes. Splitting a big data into multiple smaller chunks and placing them into multiple nodes. If we combine the data chunks from different nodes we should get back the entire data.

Why? We need to scale our system
- Scalability

If we replicate partitions into different nodes, it will ensure that the data partitions are highly available. There are 3 ways of replication ===> Single leader replication, Multi-leader replication, Leaderless replication.



In Single Leader

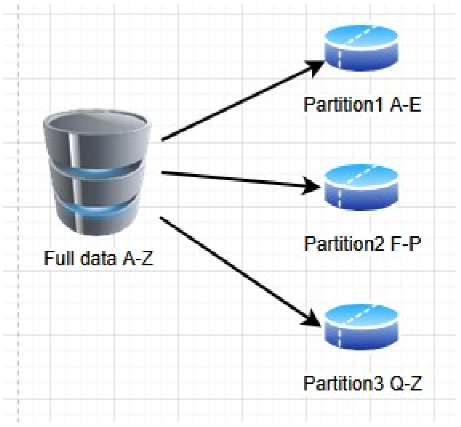


In Single leader, we have one Leader in each Node, rest are Followers. When we combine all the smaller chunks we should be able to get the full data. When the data size becomes large, we need to partition and store in different nodes.

How we are deciding which data should go where?
Partitioning of key-value data
1: {color:Red, make:BMW} is this relational data or non-relational data? Non-relational

Primary key is 1 that holds the entire value. From Primary key we can extract the entire data

We have data from A-Z, we are partitioning data into 3 parts: A-E, F-P, Q-Z. We are dividing values based on the key-values.



- No key should be left
- After partitioning, we don't have a central DB
- In each partition, the keys are stored
- Another system should know what key range and where we are storing the partition

When we do partitioning, we want to distribute data evenly across multiple partitions. If data is not evenly distributed, then one partition is likely to take most of the queries.

The partition that has most of the data is called as Skewed partition or Hot spot and the partition that doesn’t have any data is called a Not used partition

In order to cure this, we have to assign in such a way that no Hotspots are created.

Solution:
- Attach a timestamp. While performing partitions, we will attach timestamps to them. We attach timestamp to every query to identify which partition is receiving heavy traffic and accordingly we divide data into different partitions.
- Partitioning by hash of key. To avoid skew and hotspot. Eg: Cassandra, MongoDB. They use the hash of the key to perform partitioning. MD5 is a kind of hash function. Assigning partition to a range of Hash. No need of using complex algorithms. Pick hashing algorithm smartly. If there is Hash collision, we can combine multiple fields to it – add some prefix, add some postfix and re-create Hash again. Read operation: ID => Hashed => Access proper partition
- Partitioning secondary index:
Part1:
Say in our database, we have a table Cars (ID, Make, Color, Price). Say if we get most of the queries on the field: Color, we index on that field: Color. We create a lookup on Color.

What’s a Secondary index?
Partition1:
Primary key index:
101 : {color: Blue, make: BMW}
205: {color: Red, make: Audi}

Partition2:
Primary key index:
206: {color: Blue, make: Toyota}
218: {color: Black, make: Ford}

Since most of the queries is for color, we create a Secondary index

Partition1:
Secondary index for color:
color: Blue → 101, 206
color: Red → 205

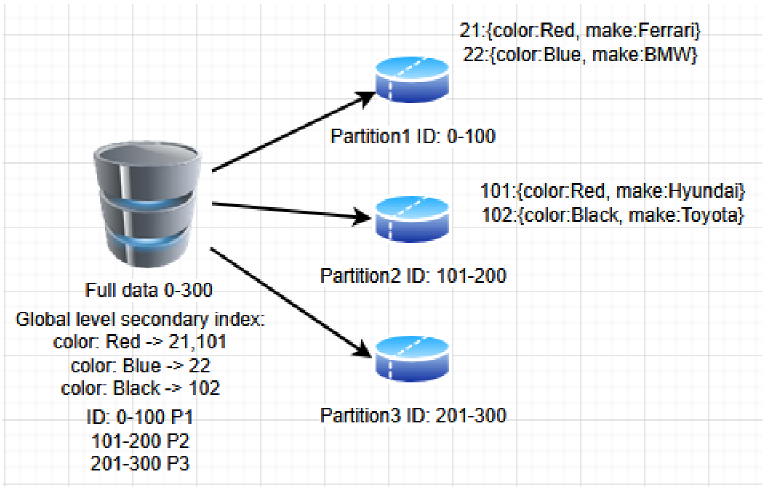
Partition2:
Secondary index for color:
color: Blue → 206
color: Black → 218

Now if we do the search, we will first find the Secondary index from here we get the database. We could also have Composite index.

Partition1	**Partition2**
Primary key index	Primary key index
101 : {color: Blue, make: BMW}	206: {color: Blue, make: Toyota}
205: {color: Red, make: Audi}	218: {color: Black, make: Ford}
Secondary key index	Secondary key index
color: Blue → 101, 301	color: Blue → 206
color: Red → 205	color: Black → 218

Say we are looking for a Blue car, we look into all partitions, in this case, it will return 101, 301 and 206
Since secondary indices are stored inside the partitions, we got to all call the partitions to access them. So the read can be expensive, time-consuming and there will be latency.
Some databases use this kind of setup: MongoDB, Cassandra, Elasticsearch
Read is slower because we don't know where the value is, we got to look into all partitions to find all Blue cars, Write is faster because we know which partition we are going to write the value into

Part2:
- By Term: Create a global secondary index so we know which partition we got to hit if we have to read data



Because we have a Global secondary index, Read will be faster in this case. Write is slower because we have to write into partition as usual plus we got to write/update into Global secondary index. Global indexes can be stored in every partition. Say a write operation is performed on P1, then we got to update other partitions as well because the Global index is stored on every partition. So write operations can be expensive. Note: Global secondary index can be handled either at Global level or Partition level.