From the vault of my engineering newsletter
"Systems That Scale"

Saurav Prateek's

Partitioning in Distributed Systems

Diving in-depth into the **Partitioning** schemes in distributed systems along with **Rebalancing** strategies

Table of Contents

Chapter 1

Partitioning Schemes - Primary Indexes

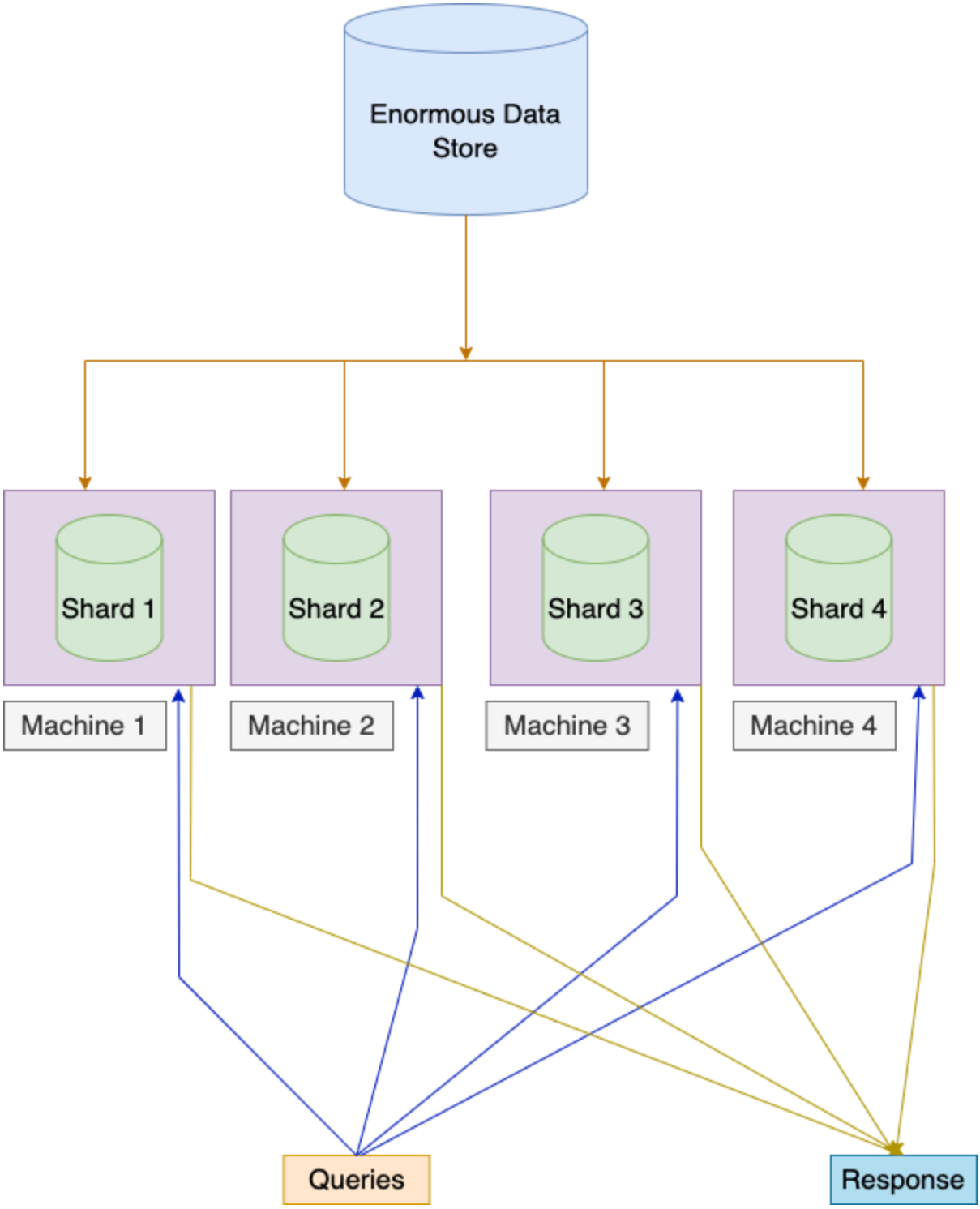
This chapter discusses how the data is actually partitioned into multiple **Shards** and what are the **Partitioning Schemes** responsible for Sharding the Databases.

Introduction

We all must have heard of **Sharding** or **Partitioning** of databases earlier as well. Even if you haven't, I will try to cover it from basics in this edition plus we will discover some interesting Partitioning schemes implemented by the Distributed Systems to partition their huge database. We will cover this topic in two editions.

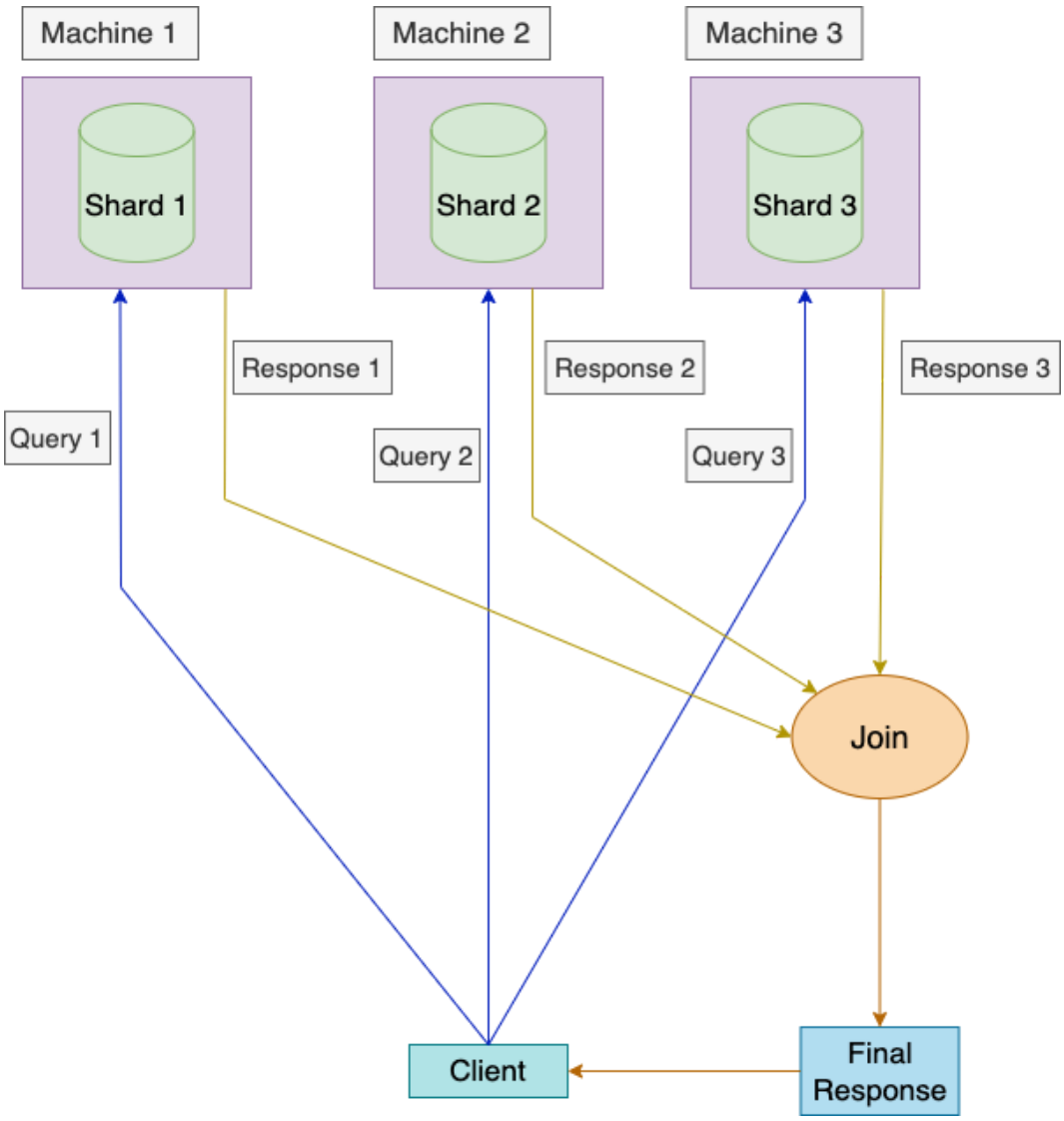
Sharding is the process of breaking up the data into partitions. This process is also known as **Partitioning**. The main idea behind sharding is to scale our systems. We can assume that each piece of data is located at exactly one shard or partition. So, every Shard behaves as an independent database of its own.

Suppose we have a large Database having an enormous amount of data. Obviously we can't store the entire data in a single server or machine. What we can do is to split the large data into smaller chunks known as Shards or Partitions and store them in independent machines. Since, now we have multiple machines holding different partitions of data, all these machines can execute queries related to them independently and in parallel. This can help in scaling the query throughput.

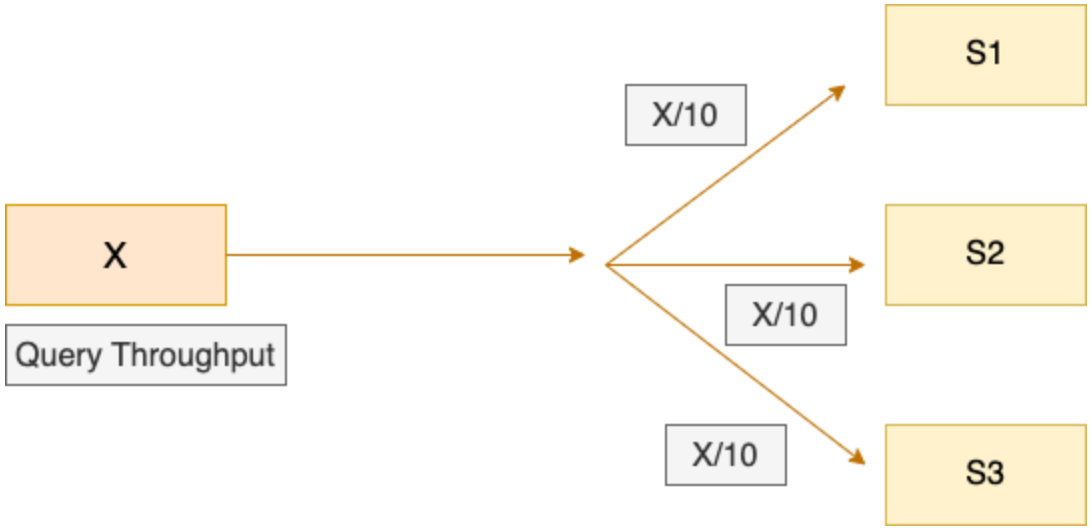


There can also be a scenario where a single query deals with data from multiple **Shards**. In that case the process might become complex. Since, we need to query

different machines holding the shards and then join back the responses received from those machines to build the final response.



Suppose we have a **key-value** datastore and we are planning to shard it. The major goal is to partition the datastore in such a way that the queries are distributed evenly across multiple shards. Suppose our system receives **X** amount of queries every hour and has **10** shards. Then ideally each shard should handle **X/10** queries. This means the system should be able to handle **10** times the load handled by a single shard.



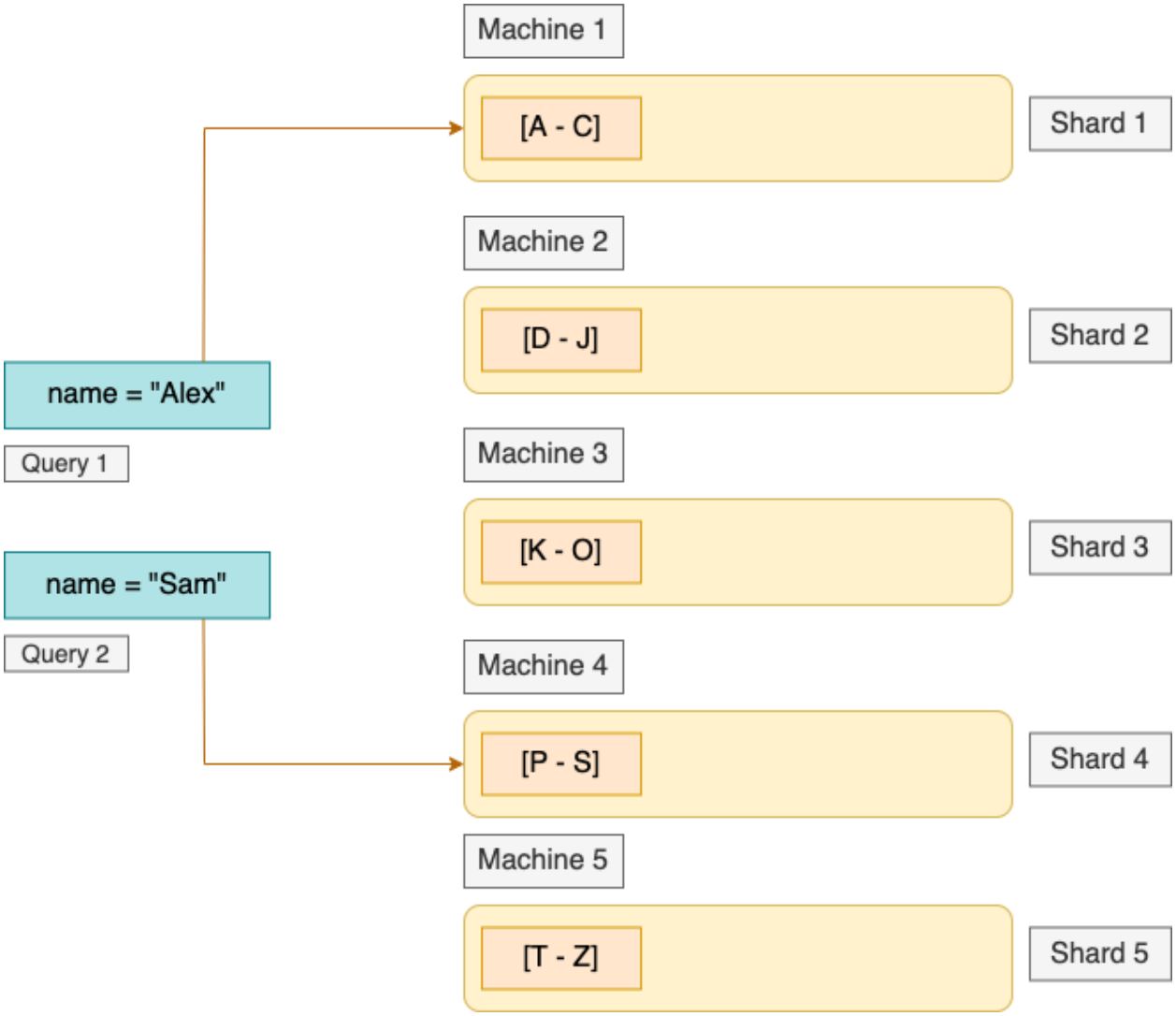
But in real life this does not happen.There is some skewness introduced in our system. If we haven't shard the data efficiently or intelligently then it might happen that the majority of queries get handled by a single shard or a minority group of shards. That single shard which handles the majority load is called a **Hot-Spot**.

Sharding by Primary Index

Suppose we have a key-value datastore where we always access a record by its Primary Key. We can shard the datastore on the basis of the Primary Index.

Sharding by Range

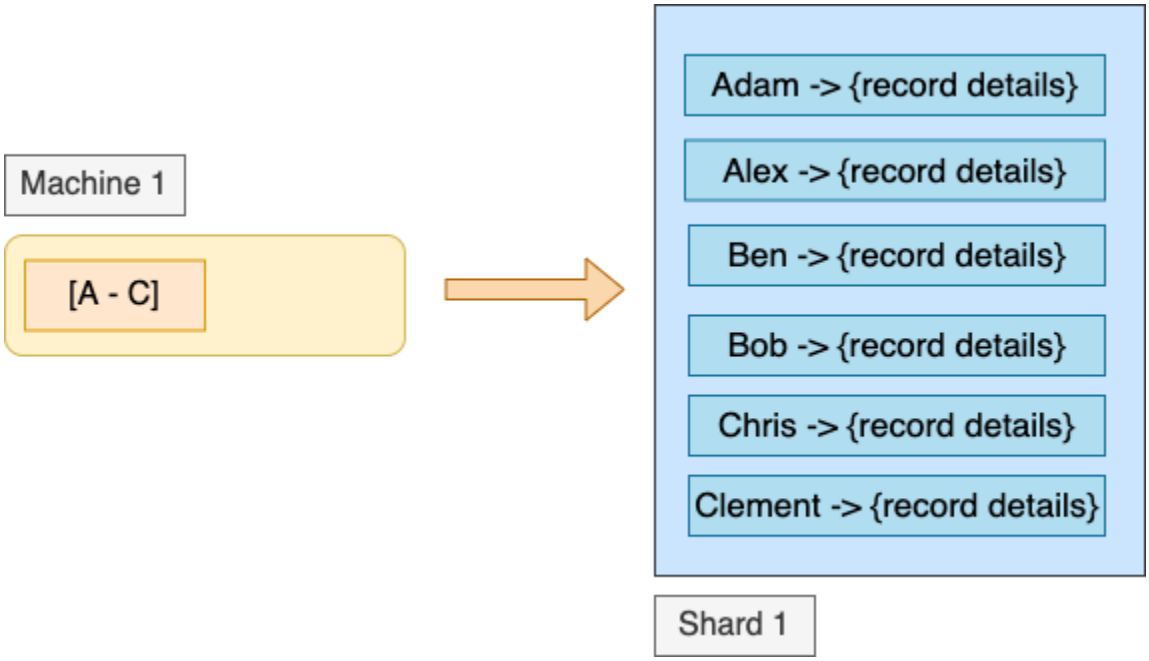
Let's assume we have a datastore consisting of the records of students enrolled for a CS course. Since we have a large number of students enrolling in the course, we have partitioned the details into different machines. Every machine stores the student details in a certain range by their names.



Every shard holds some student details. **Shard 1** holds the details of students having names starting from **A**, **B** and **C**. Hence if in future we need to query details for a student named **Alex**, we can simply send the query request to **Machine 1**.

We can observe that the range of keys are not evenly spaced. For example, **Machine 1** holds names starting with letters {**A, B, C**} while **Machine 5** holds names starting with {**T, U, V, W, X, Y, Z**}. Since the main goal is to distribute the data into partitions evenly. Hence there can be a chance that the number of students whose names start with {A, B, C} might be equivalent to the number of students whose names start with {T, U, V, W, X, Y, Z}.

Moreover, within each shard we can keep the records of students sorted by their names. This will help in reducing the lookup time in a shard. This also makes the range queries much more efficient.



Hotspots in Range based Sharding

One of the major drawbacks of the Range based partitioning is that some access patterns can lead to **hotspots**.

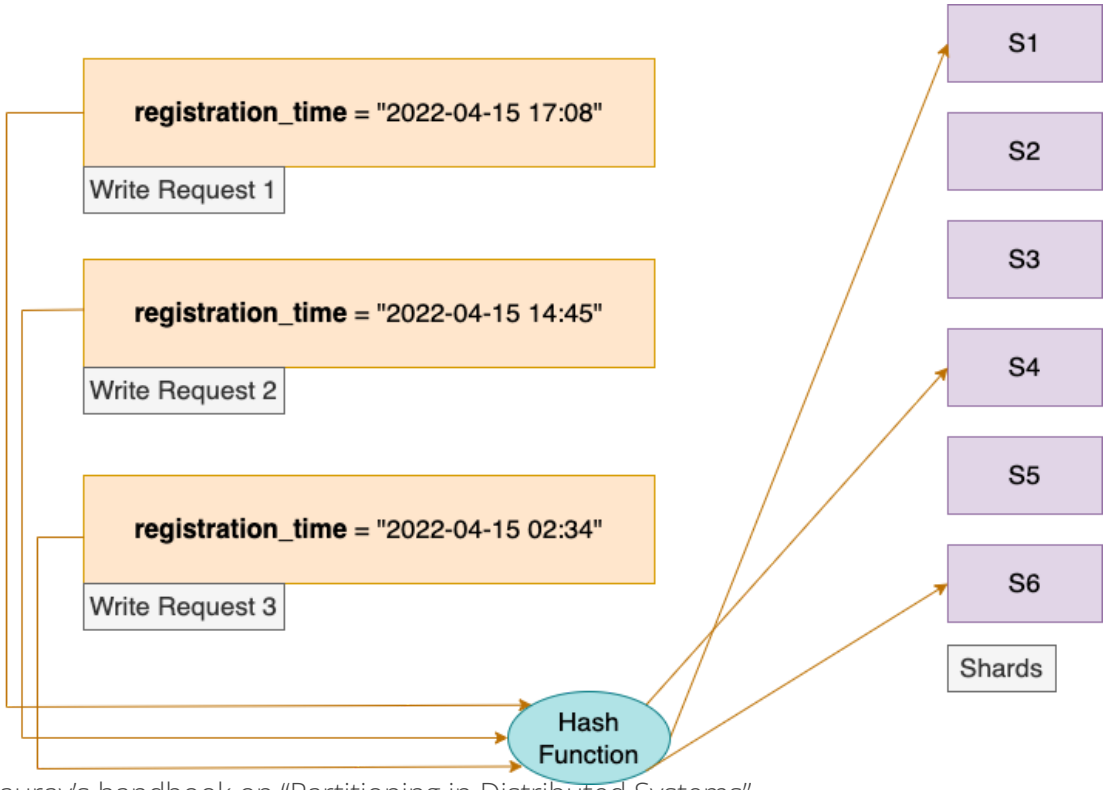
Let's take our previous data store of students as an example. This time the student records have a different Primary index. We are storing them in shards on the basis of the **Timestamp** when they take up the course. Let's say the students registering for the course on Day-1 are being stored in the 1st shard. In this way we can have a distribution of 1 shard per day. Now suppose on a certain day there was a discount on the course and a large number of students signed up on that particular day. In this case, one shard will be handling a huge number of writes on that day while the rest of the shards sit idle.

Let's see how Hash based sharding can help in avoiding this issue of Hotspots.

Sharding by Hash

We previously saw the problem of **Skewed Load** when we partitioned the data by range on **Primary Key**. To avoid this we can use a **Hash Function** in order to determine the partition of a given key.

Now the Hash function will evenly and randomly distribute the records to the shards. In our previous access pattern we stored the records of students signing up on one day on a particular shard. Now with Hashing in picture the students registering up on the same day will be sent to different shards. Since the registration time is used as a Primary Index here, the primary index value is passed through the hash function and the converted hashed value is further sent to the shard. Although the date of registration is the same, the registration timestamp is different for all the write requests and hence different hash values will be generated by the hash function. This will avoid the existence of Hot-Spots in our architecture.

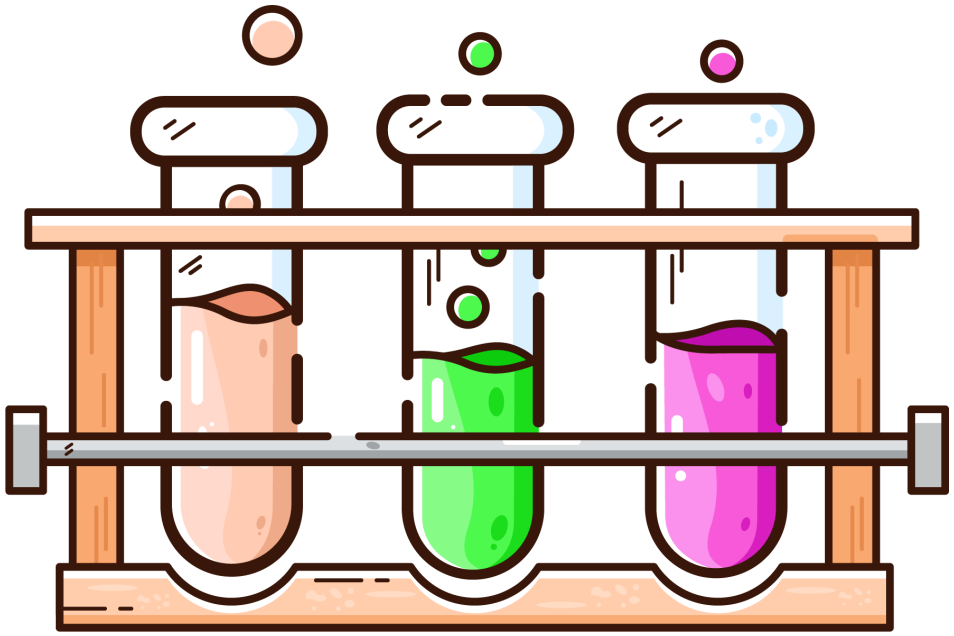


There is one major drawback of distributing the records by Hash of key. We lose the ability to perform efficient Range-based queries. Earlier the records which were stored adjacent to each other are now distributed across the shards. In this situation our range query might be sent to multiple shards out there. Then we will further need to join the responses from all the shards to build the final response.

Chapter 2

Partitioning Schemes - Secondary Indexes

This chapter discusses the Partitioning Schemes in Databases that involve **Secondary Indexes**. In this chapter we will look around the Partitioning Schemes that deal with Secondary Indexes and will also explore the concept of **Local** and **Global Secondary Indexes** as well.



Introduction

In the previous chapter we discussed the Partitioning Schemes on a key-value pair datastore where the records were only accessed through their **Primary Index**. Now things will become more interesting when **Secondary Indexes** are introduced in the partitioning process.

"Secondary Indexes does not identify a record uniquely but rather is a way of searching for occurrences of a particular value"

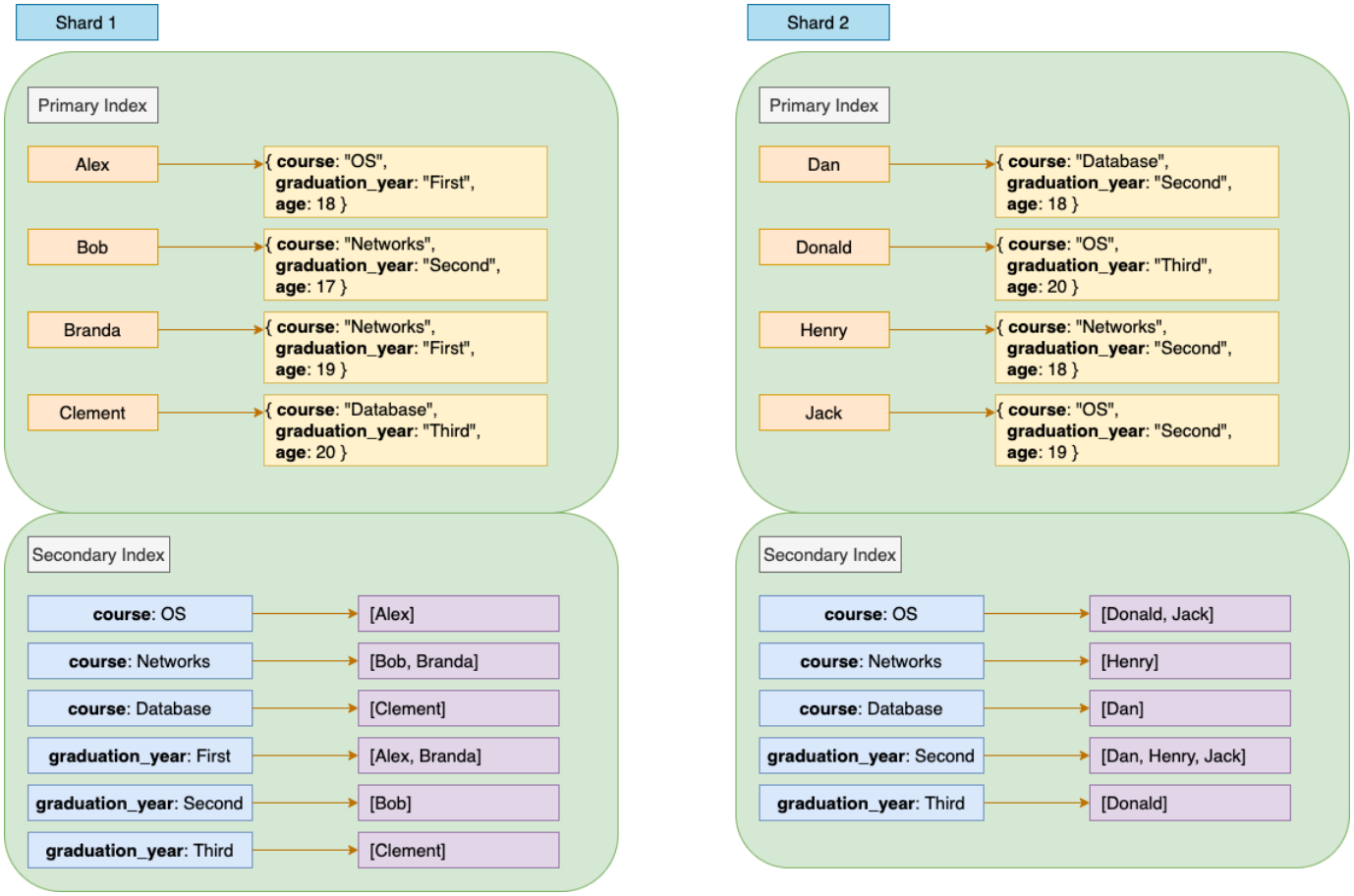
There are two ways by which we partition a data-store with Secondary Indexes.

Sharding by Document

In this partitioning scheme every partition is completely independent. Each partition maintains its own set of secondary indexes. So whenever we need to write a record we only need to update the content of the partition/shard that deals with the Primary Index of the record we are writing.

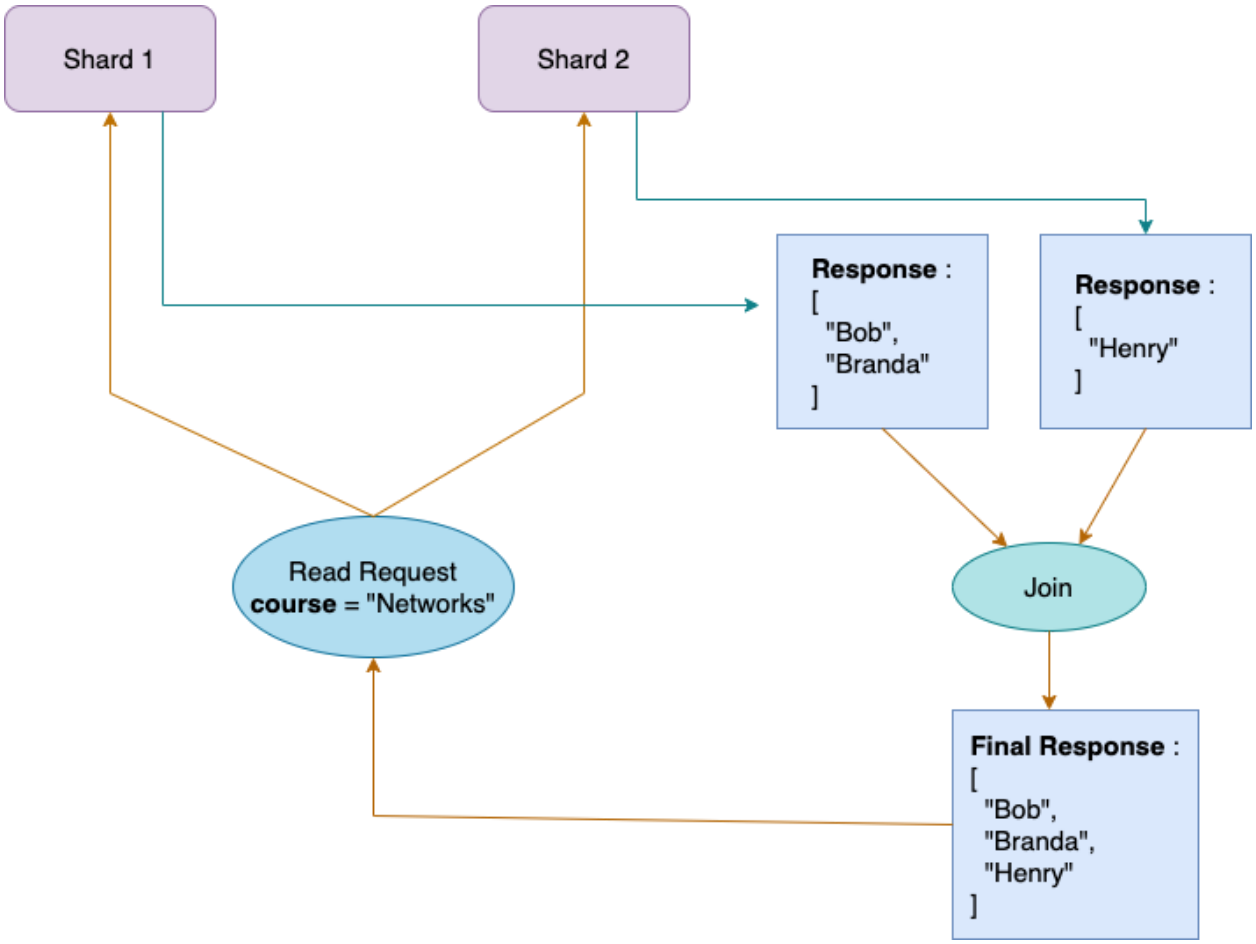
Let's take the previous data-model (the data-model we used in Part-1) of Student details. The Primary Index was the Name of the students. Now apart from the name of the students, we also need to search the records on the basis of the **Course** taken up by the students and the **Graduation Year** of the students. Hence our current data-model has Primary Index as name and **Secondary Index** as Course and Graduation Year.

Since every partition maintains their own secondary indexes we can also call them as **Local Indexes** or **Local Secondary Indexes**. Now our data-store will look like this.



In the above data-model we can observe that every Shard has maintained its own secondary indexes. Example: Both **Shard-1** and **Shard-2** have course: **Networks** as a part of their secondary index. A particular **Secondary Index** of each shard holds the primary indexes of those records which are present in that particular shard and are also associated with that secondary index.

Querying or Reading from a **Document-Partitioned** secondary index can be complex and require some extra processing. Suppose we want all the records of students who signed up for Course: **Networks**, then we will be required to query all the shards present in the system. When we get the individual responses from every shard then we combine them and return the final response. This process of querying a partitioned database is known as **Scatter/Gather** approach.



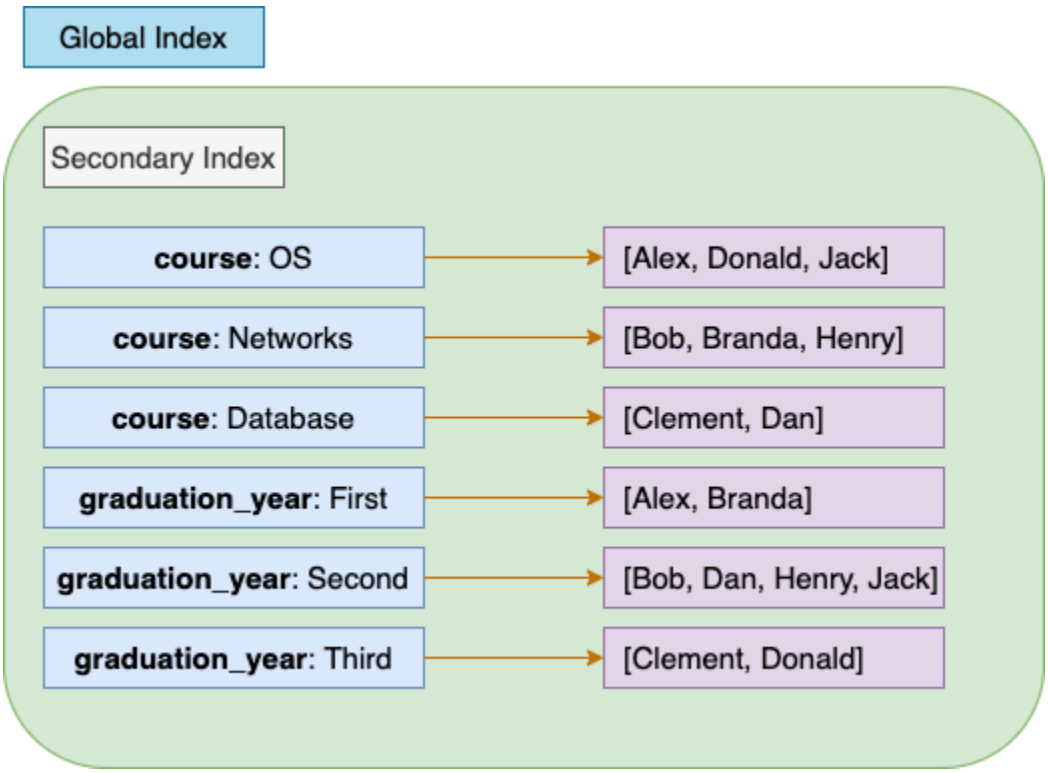
Writing to the **Document-Partitioned** indexes is fairly simple. Since every shard/partition maintains its own local secondary indexes, hence writing a record will require changes in a single partition only. The local secondary indexes maintained by the partition responsible for storing the new record will only be updated.

Sharding by Term

In the previous Partitioning Scheme every partition/shard maintained their own **Secondary Index** which can also be termed as Local Index. In this approach we won't be having a concept of **Local Index** instead we will be dealing with **Global Index**.

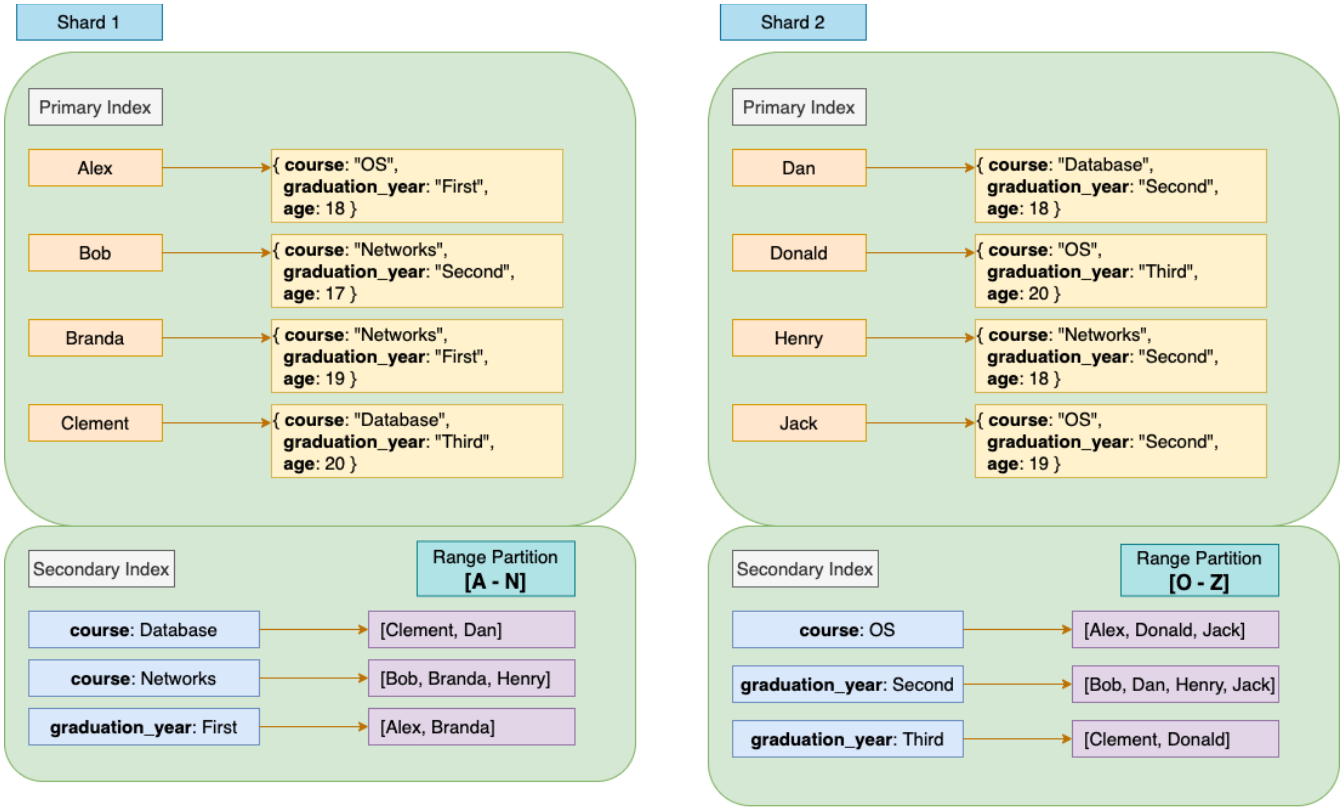
By Global Index we meant that one secondary index will hold all the keys related to that index (from all the shards). We can simply query the term we are looking for and get all the results from a single partition.

The Global Index or **Global Secondary Index** for the data-model will look like this.



The **Global Secondary Index** looks like this. We can observe that every secondary index holds the values (Primary Indexes of records) from all the partitions. Hence we call them global. Now, this Global Index alone can be huge and must be partitioned.

We can perform a range partitioning of the Global Index. The first partition can hold all the Secondary Indexes which start from letters **{A to N}** and the second partition can hold the indexes which start from letters **{O to Z}**. After partitioning the Global Index, our data-model will look like this.



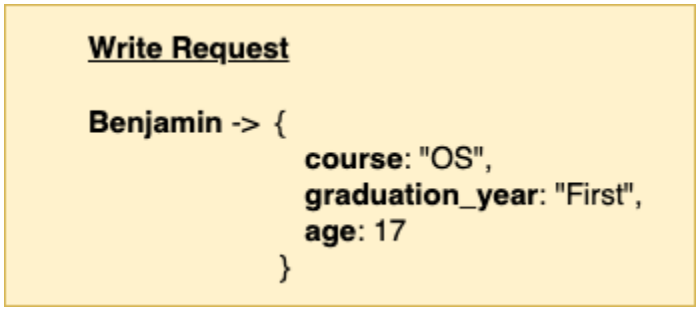
In the above data-model we partitioned the **Global Index** through index range. We can also perform **Hash-based** partitioning which can give more even distribution of the load. By **Range partitioning** we can perform Range queries efficiently.

One advantage of **Term-partitioned** index over a **Document-partitioned** index is that the reads are efficient in Term-Partitioned data model. We need to find the partition that contains the term (Secondary Index) and query that partition. Whereas in a Document-partitioned scheme we need to perform a **Scatter/Gather** approach and combine the responses from all the partitions.

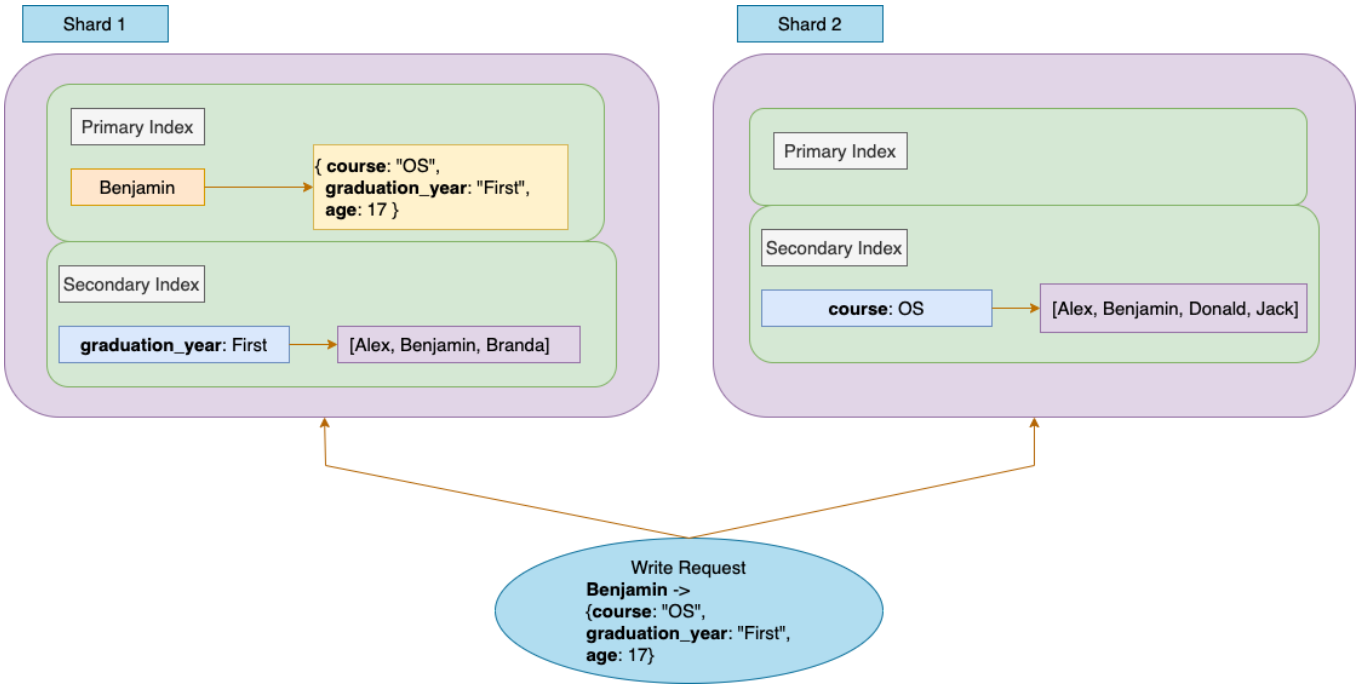
Suppose we need to query all the students who opted for a **Networks** course. Then we can simply look for the partition that holds the term "**Networks**". In our case which is **Partition-1**. We can further query **Partition-1** to get the keys of all the students who opted for **Networks** course.

One drawback of the **Term-partitioned** index over the **Document-partitioned** index is that the **Writes** are complex and slow. Writing a single record might involve

changes to be made in multiple partitions. Suppose we need to write the following Student record to the Data-model.



Writing this record will require us to update both the partitions.



The **Global Secondary Index** maintained by the partitions might take some time to get updated once the new record is added, since the process is often **asynchronous**.

Chapter 3

Rebalancing the Partitions - A naive approach

The chapter discusses the concept of **Rebalancing** in detail and a **Hash-modulo** based strategy along with its major drawback.

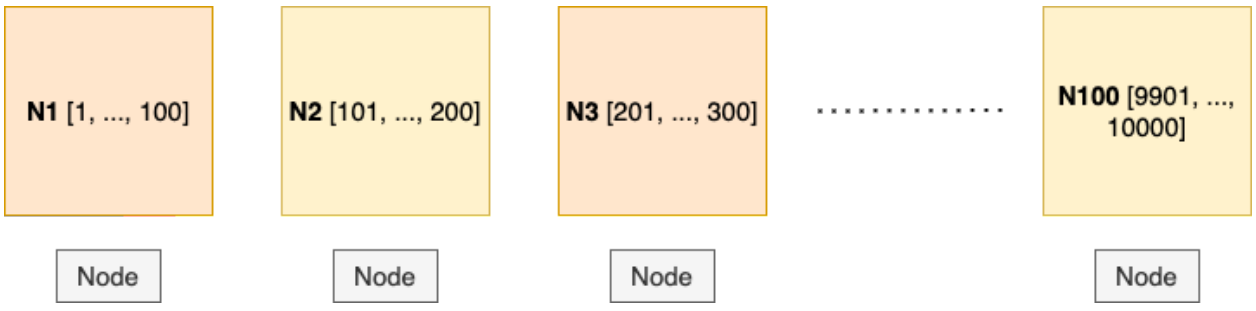
Introduction

In our previous editions we discussed **Partitioning** and multiple Partitioning schemes as well. A single database is partitioned into multiple nodes and every data-item of the database is present in at least one of the nodes. Until things are running smoothly this scheme looks good. But things might change in a database or among the nodes/machines holding them.

- What if the size of the database starts growing fast and hence we need to add more disks to store the data.
- What if the load on the system increases and we need to add an extra machine to scale our system.
- What if an existing machine/node crashes due to some unforeseen hardware issue.

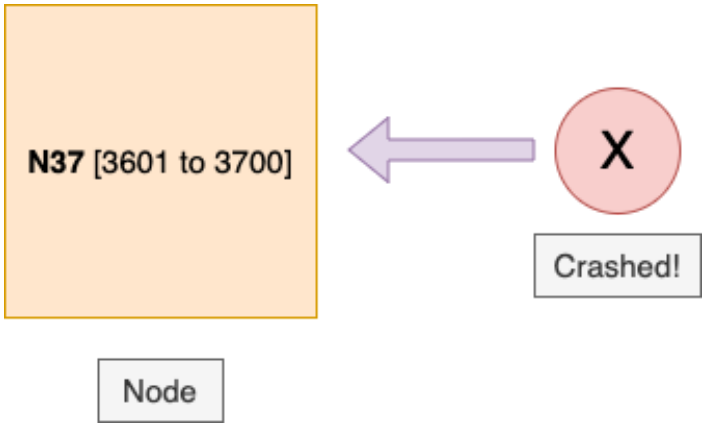
All of the above changes will require the existing data-items to move across the nodes. This process of moving a data-item from one node to another is called **Rebalancing**.

Let's take a fairly simple example to understand this. Suppose we have a database holding about **10,000** data items which are partitioned over **100** nodes such that every node holds about **100** data-items. The current architecture would look like this.



Scenario 1

Now imagine one of our nodes says **Node-37** crashes due to some hardware failure. Ideally we would want all the data-items present in the Node 37 to be still available to the users. In order to do so we need to immediately re-distribute the data-items present in that node (failed node) to the rest of the available (healthy) nodes.

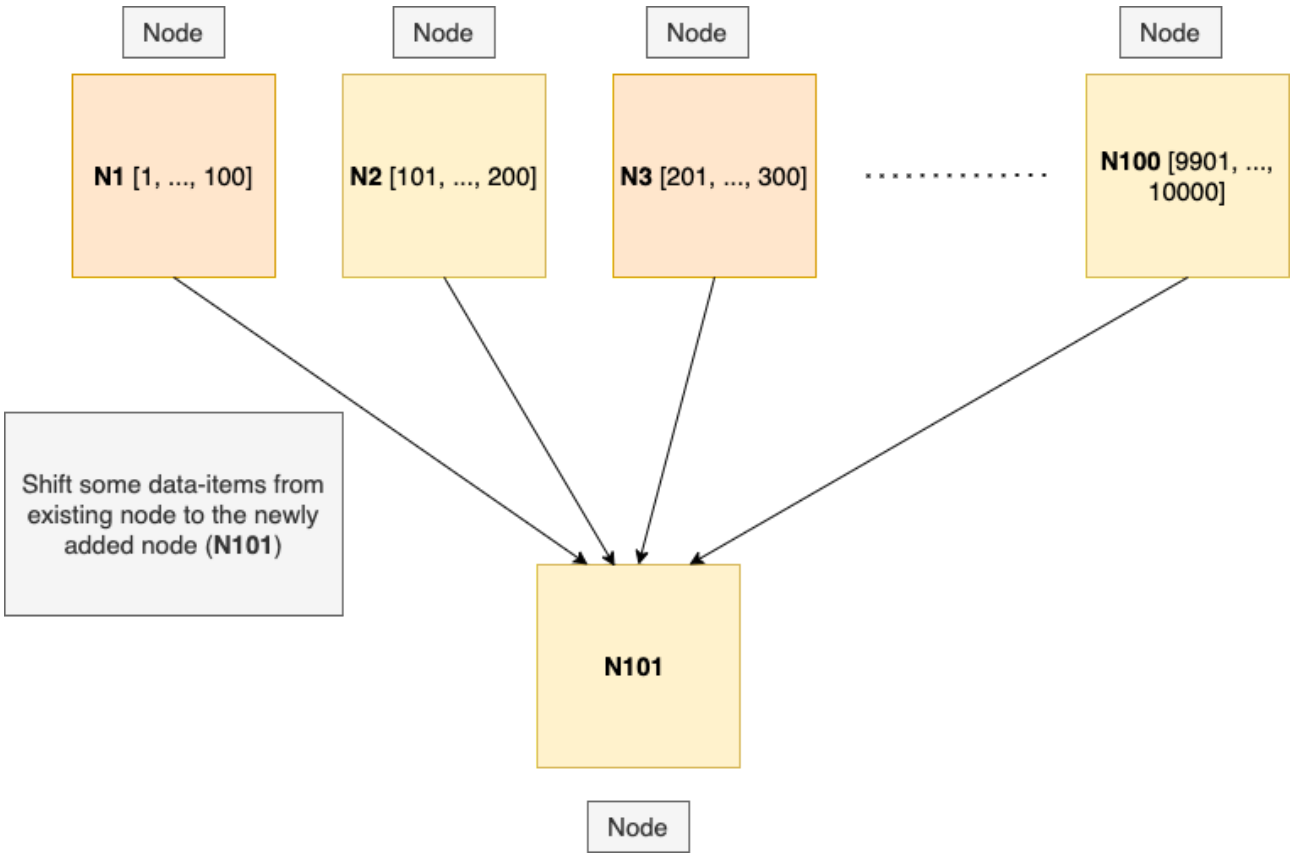


In the above node we can see that data items with Ids **3601** to **3700** need to be shifted to other available nodes. We could try distributing these data-items to the existing node in such a way that the amount of data-items held by every node is almost even. We are aiming for an even distribution of data-items across the available nodes. There are multiple schemes to perform this rebalancing which we will discuss in the upcoming sections.

Scenario 2

Suppose the number of users of our system exceeded and now we are planning to add one more node/machine to our cluster to distribute the incoming queries. Since our queries throughput increased we are planning to scale our system. We had **100** nodes previously and now added an extra node **Node-101** to the existing

cluster. Now we also need to put some data-items from the existing nodes to the newly added node.



We looked at two different scenarios where in one of them we removed an existing node while in the other we added a new node into the existing cluster of nodes. During the rebalancing process these are three requirements which are needed to be fulfilled.

1. **Requirement 1**: After rebalancing, the load (data-items present in the nodes, write and read requests) must be evenly shared among the existing nodes in the cluster.

2. **Requirement 2**: While rebalancing only those data-items which are required to be shifted across the nodes should be moved. This saves network and disk

I/O load. Remember in the Scenario-1, we moved only the data-items present in the crashed node to the other healthy nodes.

3. **Requirement 3**: During the rebalancing the database must be available i.e. it should continue to accept reads and writes.

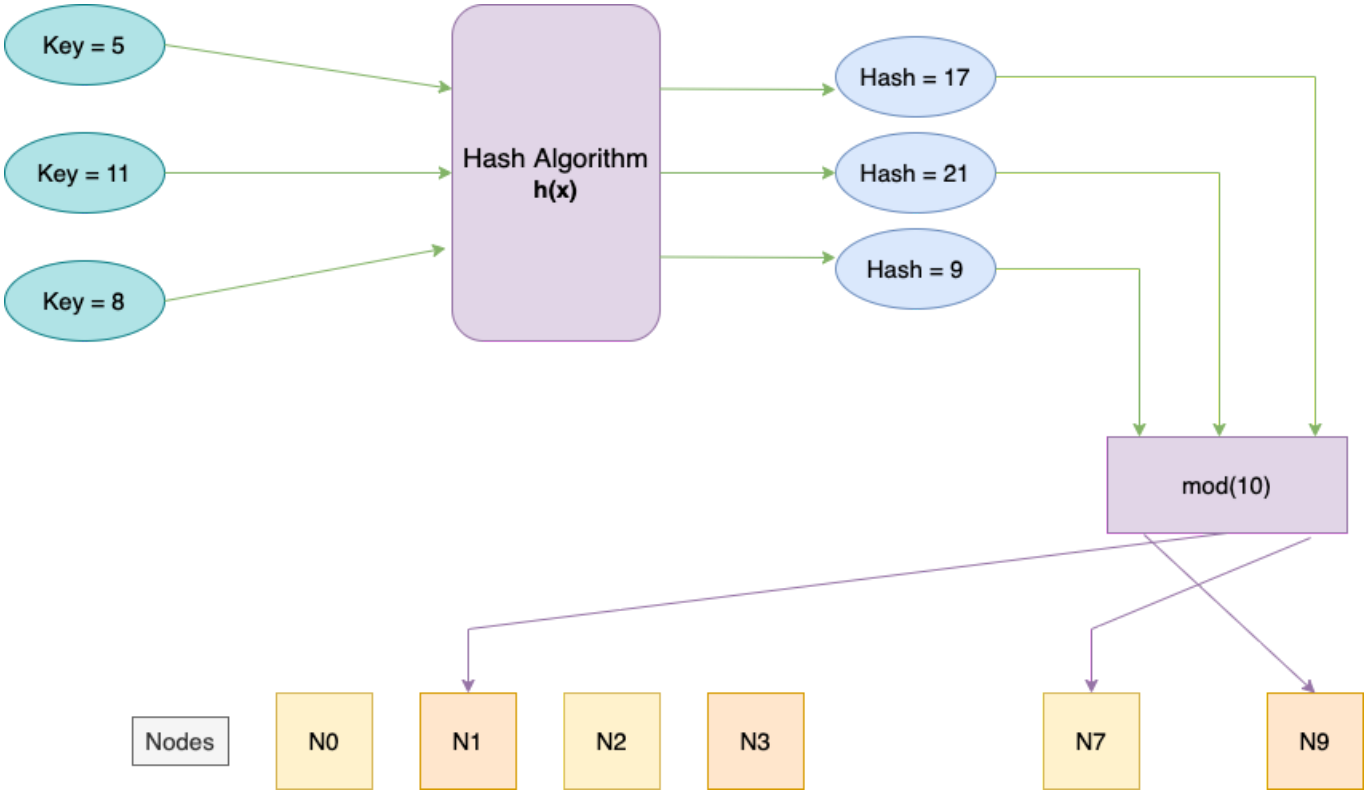
Strategies for Rebalancing

There are multiple strategies under which we can perform rebalancing while delivering the above requirements as well. We will probably discuss these strategies in the next edition. But let's look upon an interesting strategy which we should probably avoid while rebalancing.

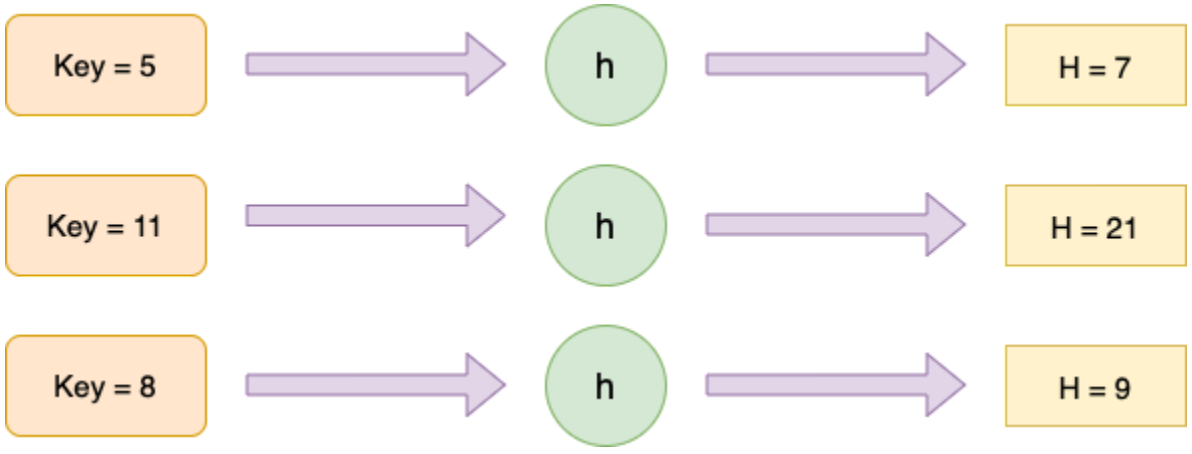
Performing Hash-Modulo (Should be avoided)

Under this strategy we partition by the Hash of a key. We use a **Hashing Algorithm** that generates random hash values after taking an input as a number. After that we modulo the resultant hash-value with the total number of nodes present in the cluster to randomly assign it to one of them. This method although distributes the data-item evenly across the existing cluster of nodes but still this method should be avoided. We will discuss its drawbacks ahead.

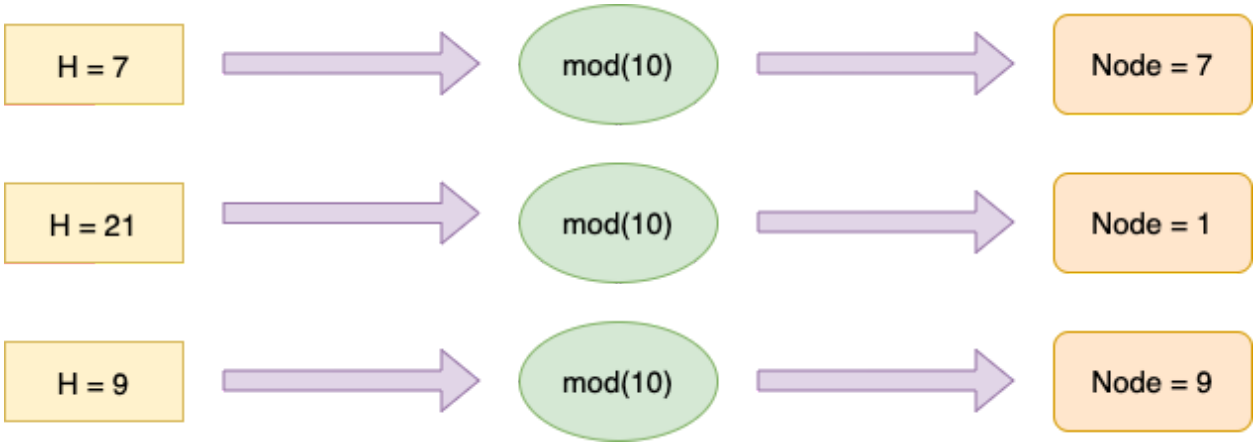
Suppose we have **10** nodes present in a cluster. We also have a Hashing Algorithm **'h'** that can generate random hash values by taking a number as an input. The entire hash-modulo strategy looks like this.



The hash function (**h**) generates following random hash values for the input keys.



Now we perform the modulo **10** on the resultant hash value to get the node ID to which they will be directed to.



So far this scheme looks good since it's able to distribute the incoming keys to the nodes evenly. Now, what would happen if we perform rebalancing by increasing or decreasing the number of nodes in the cluster.

Let's observe the scenario when the number of nodes in the cluster is decreased from **10** to **9**.

Key	Hash	Hash Value	Mod	New Node Address	Old Node Address
5	h	17	mod(9)	8	7
11	h	21	mod(9)	3	1
8	h	9	mod(9)	0	9

Let's also observe the scenario when the number of nodes in the cluster is increased from **10** to **11**.

Key	Hash	Hash Value	Mod	New Node Address	Old Node Address
5	h	17	mod(11)	6	7
11	h	21	mod(11)	10	1
8	h	9	mod(11)	9	9

In both the above cases the new address for almost all the keys are different from the earlier node's address. Even though we added or removed one node from the cluster it led to the transfer of almost all the keys to different nodes.

This contradicts with the second requirement of rebalancing in the partitioning scheme which states that only the necessary data-items should be moved across the nodes while rebalancing. Such frequent moves can be expensive and hence we avoid this scheme.

Chapter 4

Strategies for Rebalancing the partitions

The chapter discusses the **Fixed** and **Dynamic** partitioning as the strategies for Rebalancing along with their advantages and pitfalls.

Introduction

In our previous chapter we discussed **Rebalancing** and the challenges which require us to introduce it in our systems. We also discussed one strategy that was not very efficient for performing rebalancing. Do check that out as well since it can be a necessary prerequisite for this edition.

In this chapter we will look around some **strategies** for rebalancing which are efficient and also used by multiple database services now-a-days.

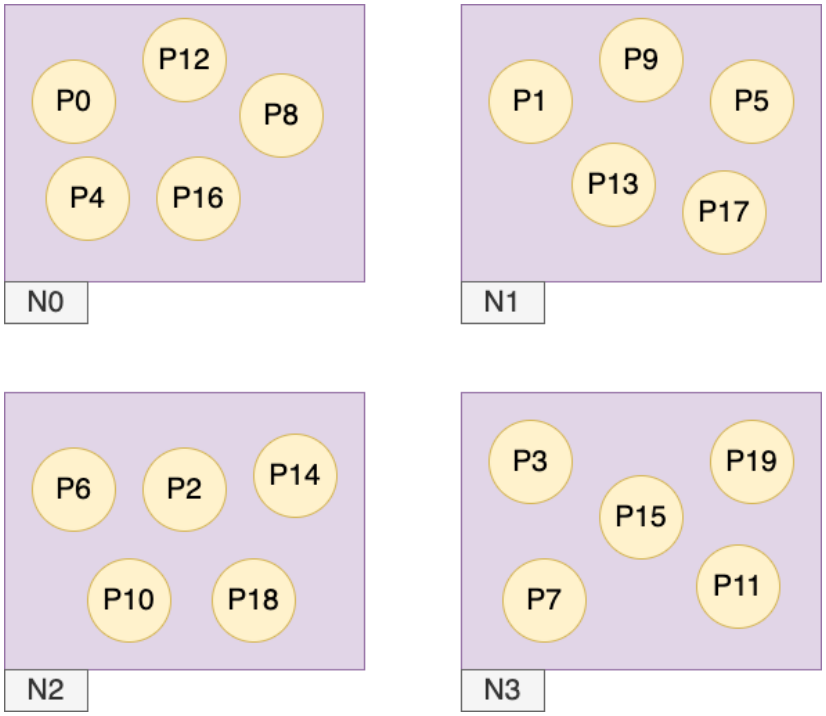
Fixed Partitioning

In our previous edition we discussed a strategy which was quiet in-efficient and involved very frequent transfer of data across nodes. To solve the problem, in this scheme we can split our database into a large number of partitions and then assign multiple partitions to each node. For this, the number of partitions must be many more than the number of existing nodes in the system.

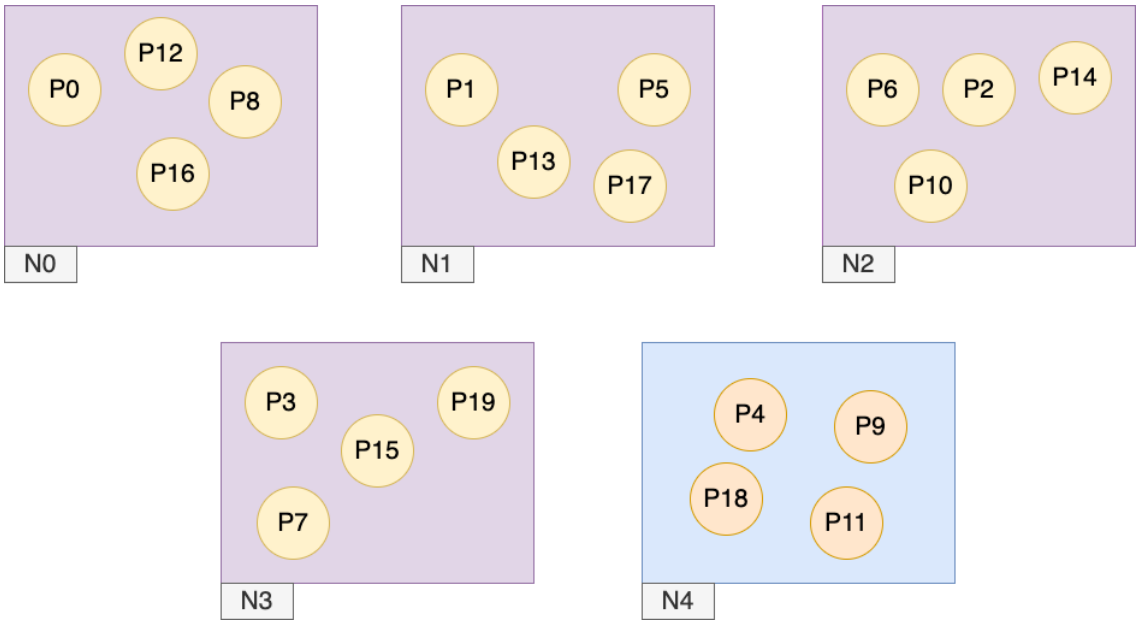
Suppose our database has **N** nodes, then we can split our database into **10N** partitions and further assign **10** partitions to every single node.

In this configuration the number of partitions remains constant and the assignment of keys to the partition also remains the same. The only thing that changes is the assignment of partitions to the nodes.

Let's take a look at this strategy by an example. Suppose our database initially had **4** Nodes { **N0, N1, N2, N3** }. The database has been split into **20** partitions { **P0, P1, ... , P19** } which will be fixed throughout. Hence the initial configuration looks somewhat like this.

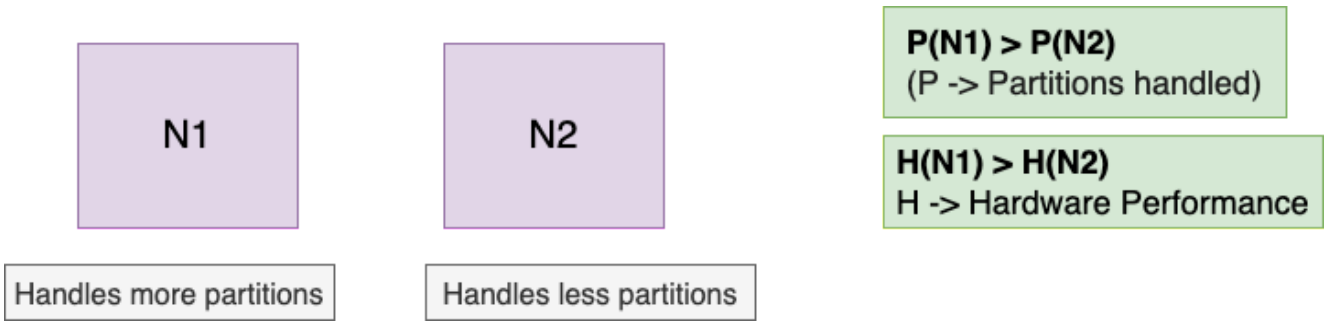


Now, a new node **N4** is introduced in our system. This new introduction will cause multiple existing partitions to be re-assigned to this new node. This will cause some change and our new configuration will look like this.



Nodes with variable Hardware Performance

This scheme allows the system to utilise the nodes according to their hardware capacity. Suppose node N1 has a higher hardware performance than N2 then we can assign more partitions to node N1 as compared to N2. More powerful nodes can take a greater share of load.

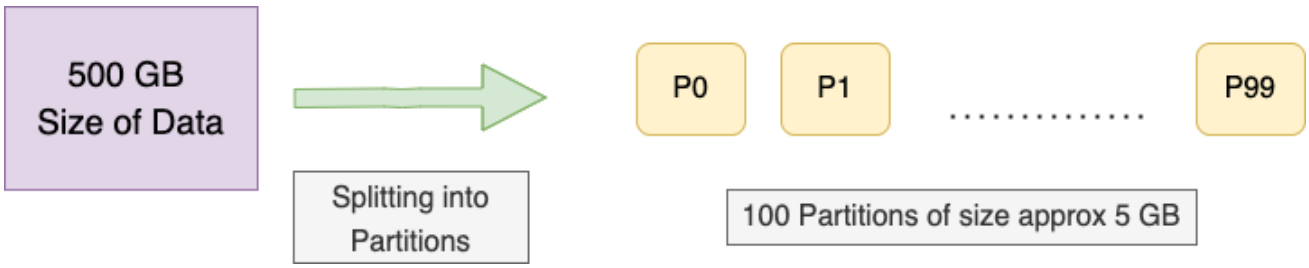


Choosing the right number of Partitions

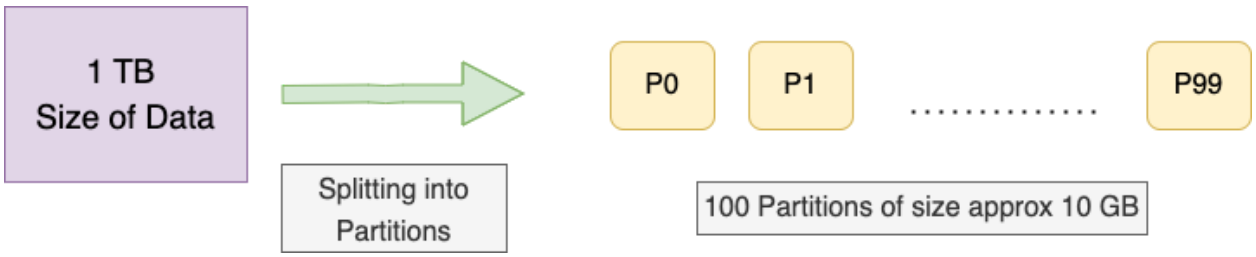
Since in this scheme the number of partitions is fixed at the start and can not be changed afterwards. This puts a pressure on choosing a correct number of partitions initially.

As the number of partitions is fixed hence, as the total amount of data grows the size of the partitions grows proportionally. Hence, it's difficult to choose the right number of partitions initially when the total size of the data in the system tends to be highly variable.

Suppose the size of the total amount of data in the system is **500 GB** (approx.) and the number of partitions decided initially is **100** then each partition would be of size approx. **5 GB**.



In future if the total amount of data in the system doubles down to **1 TB** (approx) then the size of each partition will grow around **10 GB**.



- If the number of partitions are very large then re-balancing and recovery from node failure becomes hard and expensive.
- If the number of partitions are very small then they can incur too much overhead.

The best performance is achieved when the number of partitions is just right, neither too big nor too small.

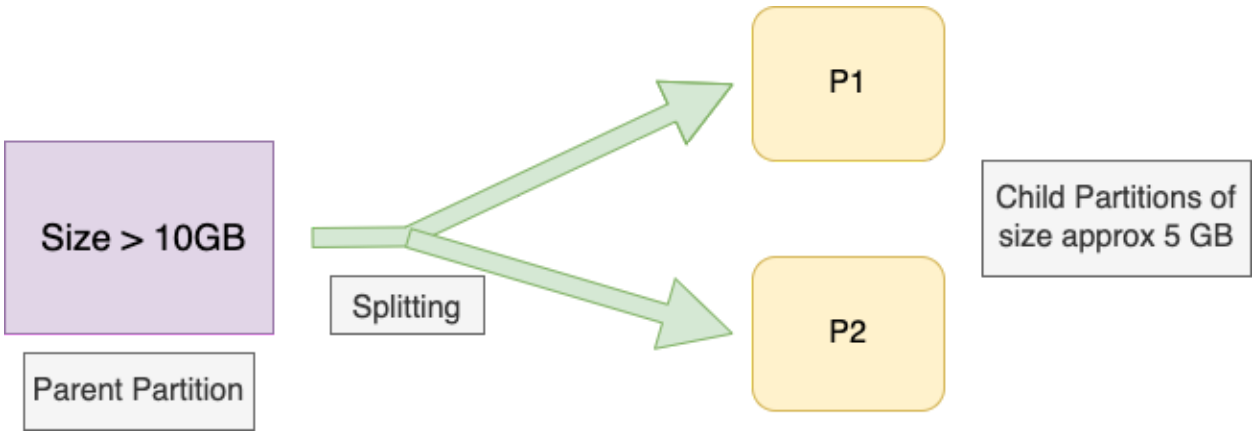
Dynamic Partitioning

In the previous scheme of **Fixed Partitioning** there was a difficulty of choosing the number of partitions initially, especially for those systems whose total amount of data is highly variable.

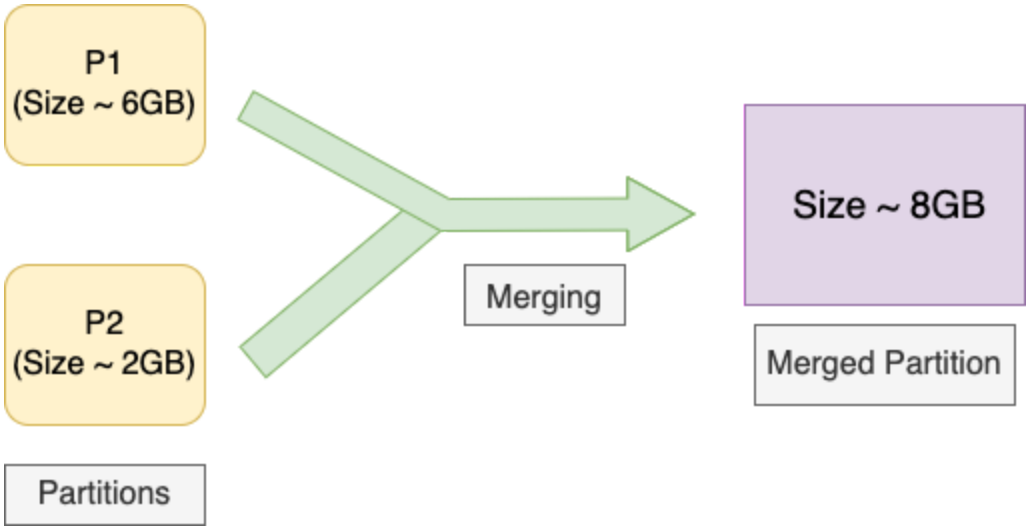
In this scheme the number of partitions is variable and adapts to the amount of the total data volume.

Splitting and Merging the Partitions

Many databases maintain a threshold value which if a partition's size exceeds then that partition splits into two. **HBase** has a threshold value of **10 GB**. Suppose a partition exceeds the size of **10 GB** then it is further split into two partitions of approx. **5 GB** each. After splitting, one of the halves can be transferred to another node in order to balance the load.



Similarly, if lots of data is deleted from a partition and it shrinks below the threshold, then it can be merged with an adjacent partition.



Note: Initially an empty database starts off with a single partition (empty).

Thank You! ❤️

Hope this handbook helped you in clearing out the concept of **Partitioning Schemes** and **Partition Rebalancing Strategies** in **Distributed Systems**.

Follow me on Linkedin
Do subscribe to my engineering newsletter “Systems That Scale”