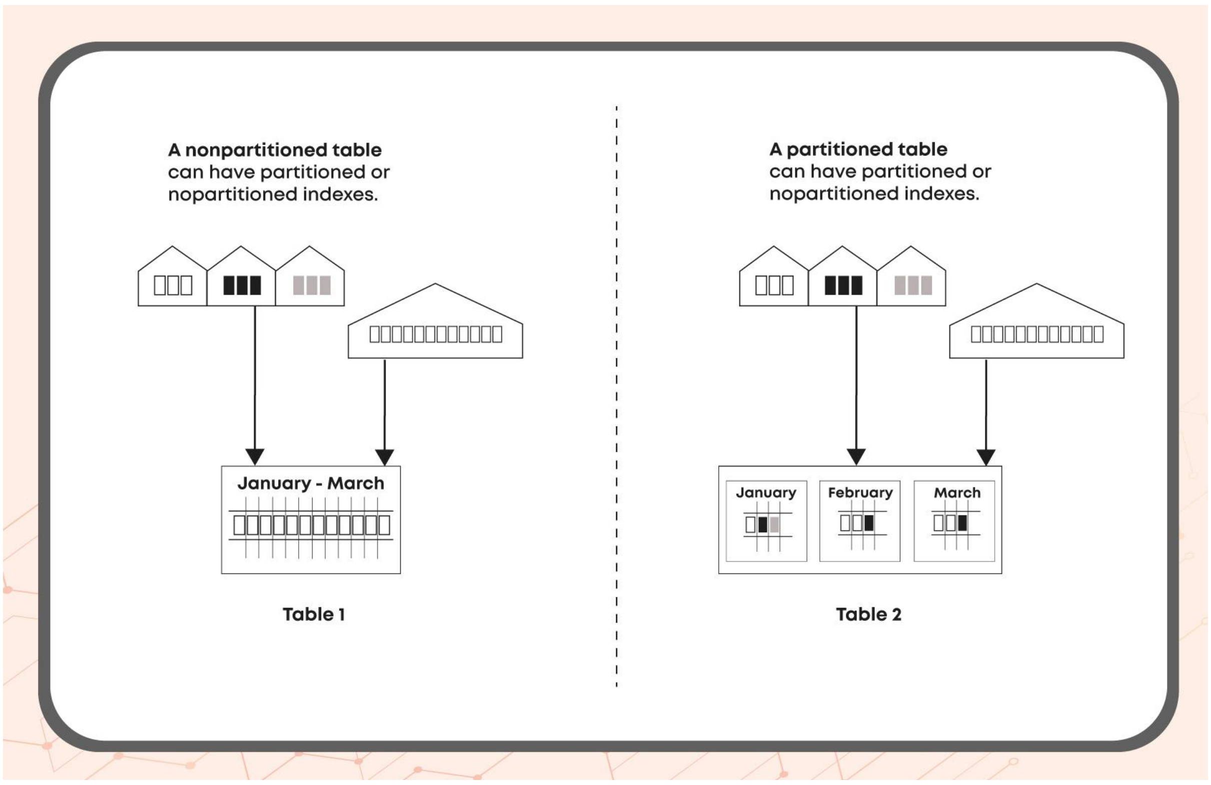
Partitioning

What is partitioning?

Partitioning is the database process of dividing a very large table into several smaller parts. By breaking large tables into smaller individual tables, queries that access only a small part of the data can run faster because there is fewer data to scan. Partitioning is done when the data is huge and a single machine cannot handle all the data.



A nonpartitioned table can have partitioned or nopartitioned indexes.
January - March
Table 1
A partitioned table can have partitioned or nopartitioned indexes.
January
February
March
Table 2

Fig: Non-Partitioned vs Partitioned Table

Advantages of partitioning

1. <u>Scalability:</u>
   For an application with a large amount of data, all the data cannot fit in a single machine. When the data increases the data must be split and put in different machines. Therefore, partitioning helps in achieving the high scalability of the application.
2. <u>Manageability:</u>

Partitioning of data makes it more manageable. Processing queries on smaller chunks of data is much more efficient.

3. Performance Optimization:
   Since the data is split into different machines. All the queries should be targeted on the same machine and can be concurrently run on all the smaller machines increasing performance and speed.
4. Availability:
   All the data is not contained in a single machine so even if a node fails or breaks down, the other nodes would be up and running and delivering data.
5. Load Balancing:
   Load is distributed over multiple machines and no single machine is overloaded now.

Disadvantages of partitioning

1. More hardware management: increased DevOps load
2. Software complexity: need to maintain the logic of splitting data, routing queries, aggregating compute results
3. More overhead when combined with redundancy

Partitioning Methods

There are multiple partitioning methods or strategies like:

Horizontal Partitioning/Shading/Range-based partitioning:

Horizontal partitioning splits large tables into smaller manageable parts without the need to create separate tables for each part. The data in a partitioned table is physically stored in row groups called partitions. Each partition can be accessed and saved separately. Each shard has the same schema as the original database in horizontal partitioning.

Example: Perhaps customers with ZIP codes less than 50,000 are stored in East Customers and customers with ZIP codes greater than or equal to 50,000 are stored in West Customers. The two partitioned tables are CustomersEast and CustomersWest, and a join view can be created on both to provide a complete view of all customers.

Challenges of horizontal partitioning:

1. Range-based:
   During the range-based partitioning if the data is not equally distributed then that may lead to an unbalanced state.
2. Increased complexity:
   Shading may increase the complexity of the system since they are located on different machines.
3. Expensive: In cases of range read queries, or queries spanning across multiple machines; we need to fetch results from all machines and aggregate results, hence can make it expensive.

Vertical partitioning:

Vertical partitioning involves creating tables with fewer columns and using additional tables to store the remaining columns. Data is broken down in the vertical style. Vertical table partitioning is mainly used to improve SQL Server performance, especially when the query retrieves all columns from a table that contains a large amount of text or BLOB columns.

Example: Suppose you have a large table containing employee reports, containing basic information such as report name, ID, number of reports, and a large column with report descriptions. Suppose that about 95% of the users are searching for the name, number, etc. of the report, and only about 5% of requests open the report description field and see the description. Assuming that all of these searches will result in a clustered index scan, and since the index scan reads all the rows in the table, the cost of the query is proportional to the total number of rows in the table. Our goal is to minimize the number of I / O operations and reduce the search cost. To reduce query costs, we will change the SQL Server database schema and split the EmployeeReports table vertically.

Challenges of vertical partitioning:

1. Slow JOIN queries:
   To join queries on different tables, the system has to make network calls to each of the tables and then join the data. The table which is partitioned on multiple machines makes it very slow. JOIN operations are required to get data from different shards which may lead to slower join queries.

2. Further partitioning required:

Sometimes further table partition is required if one particular table has grown large and it needs additional partitioning.

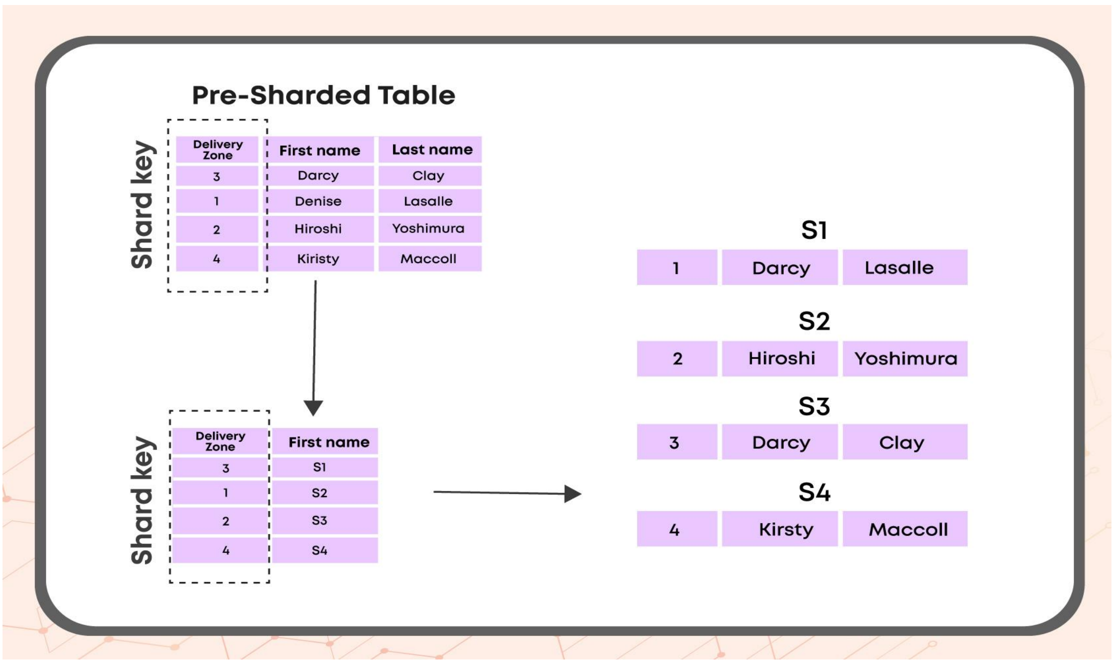
<u>**Directory-Based Partitioning:**</u>

A search service that knows the partition scheme and abstracts it from the database access code. It allows adding database servers or changing the partition scheme without affecting the application. It leads to a loosely coupled application and a horizontally scalable application. Directory-based is more flexible than the key-based or range-based partitioning as in

- key-based partitioning we need to use a hash function that cannot be modified frequently.
- In range-based partitioning, the values of ranges are decided which cannot be modified.

But, in directory-based partitioning we can use any algorithm to assign data in the shards, it is a more dynamic approach and hence more flexible.

<u>Example</u>: To use directory-based sharding, you'll need to design and maintain a lookup table that uses a shard key to track which shard contains which data. A lookup table, in a nutshell, is a table that contains a static collection of information about where specific data may be found. A simple example of directory-based sharding is shown in the diagram below:



Delivery Zone	First name	Last name
3	Darcy	Clay
1	Denise	Lasalle
2	Hiroshi	Yoshimura
4	Kiristy	Maccoll

Delivery Zone	First name
3	S1
1	S2
2	S3
4	S4

S1

1	Darcy	Lasalle

S2

2	Hiroshi	Yoshimura

S3

3	Darcy	Clay

S4

4	Kirsty	Maccoll

Challenges of Directory-Based partitioning:

1. Complex:
   This kind of partitioning is complex to implement because the directory server has a mapping of each record with their corresponding servers.
2. Single Point of Failure:
   If the directory server breaks down or fails, it would be a single point of failure scenario. Therefore, redundancy is essential in such partitioning.

Partition Criteria

1. Key or Hash-Based Partitioning:
   We apply the hash function to the key attribute of the entry to find out the partition number.

   Issues with hash-based partitioning:

   1. Dynamic adding or removing servers is difficult.
   2. During the migration of data, the hash function should be changed.
   3. Changing the hash function requires re-distribution of data.
   4. Redistribution causes server downtime.
   5. Re-distribution is also very expensive.

   Example: Partition based on the year in which a student took admission in the particular school.

```
CREATE TABLE students (
   id INT NOT NULL,
   fname VARCHAR(30),
   lname VARCHAR(30),
   admitted DATE NOT NULL DEFAULT '1970-01-01',
   separated DATE NOT NULL DEFAULT '9999-12-31',
   student_code INT,
   store_id INT
)
PARTITION BY HASH( YEAR(admitted) )
PARTITIONS 4;
```

2. List Partitioning:

The partition is selected on the basis of the column that matches one of the sets of discrete values. The particular partition is assigned a list of suitable values.

Example:

```
PARTITION BY LIST(store_id) (
   PARTITION pNorth_Delhi VALUES IN (3,5,6,9,17),
   PARTITION pEast_Delhi VALUES IN (1,2,10,11,19,20),
   PARTITION pWest_Delhi VALUES IN (4,12,13,14,18),
   PARTITION pCentral_Delhi VALUES IN (7,8,15,16)
);
```

3. Round Robin Partitioning:

   Suppose we have n partitions, then the ith tuple is assigned to partition number i%n. That means that the ith data would go to (i%n) nodes. Data is assigned in sequential order. This partitioning criterion ensures the even distribution of data.

4. Consistent Hashing:

   This is a new kind of partitioning.

   There was an issue with the hash-based partitioning that on adding the new servers, the hash function had to be changed. Changing the hash function would mean redistribution of data and server being non-available. This would be very expensive.

   In the calculation, the consistent hashing is a special kind of hashing, so when the hash table is resized, only the keys whose number of keys are and need to be redistributed on average. It is a technique that is used to build scalability into the system's storage architecture from scratch.

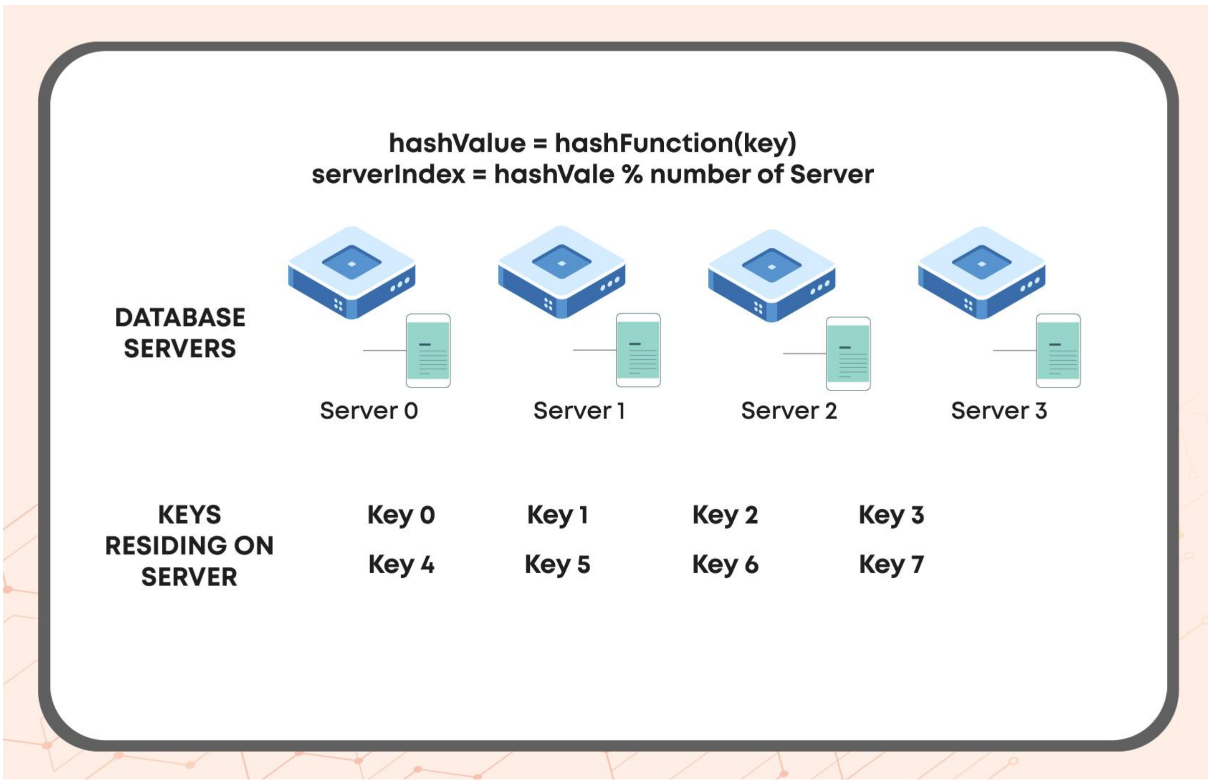
   The approach of consistent hashing is as follows:

1. hash_Value = Hash_Function(Key)
2. server_Index = hash_Value % n

Example:

- Suppose we have 4 database servers
- Suppose our hash function returns a value from 0 to 7
- We suppose that "key0" generates a hash value or 0 when passing our hash function, "key1" returns 1, etc. Etc.

- The server_Index of "key0" is 0, "key1" is 1, and so on.


hashValue = hashFunction(key)
serverIndex = hashVale % number of Server
DATABASE SERVERS
Server 0
Server 1
Server 2
Server 3
KEYS RESIDING ON SERVER
Key 0
Key 4
Key 1
Key 5
Key 2
Key 6
Key 3
Key 7