Database Scaling

Before starting with different ways to scale databases, let's first understand the problem statement.

Problem Statement

Imagine you have a customers table with 10 billion rows. Querying such a large table can be very slow. Here's why:

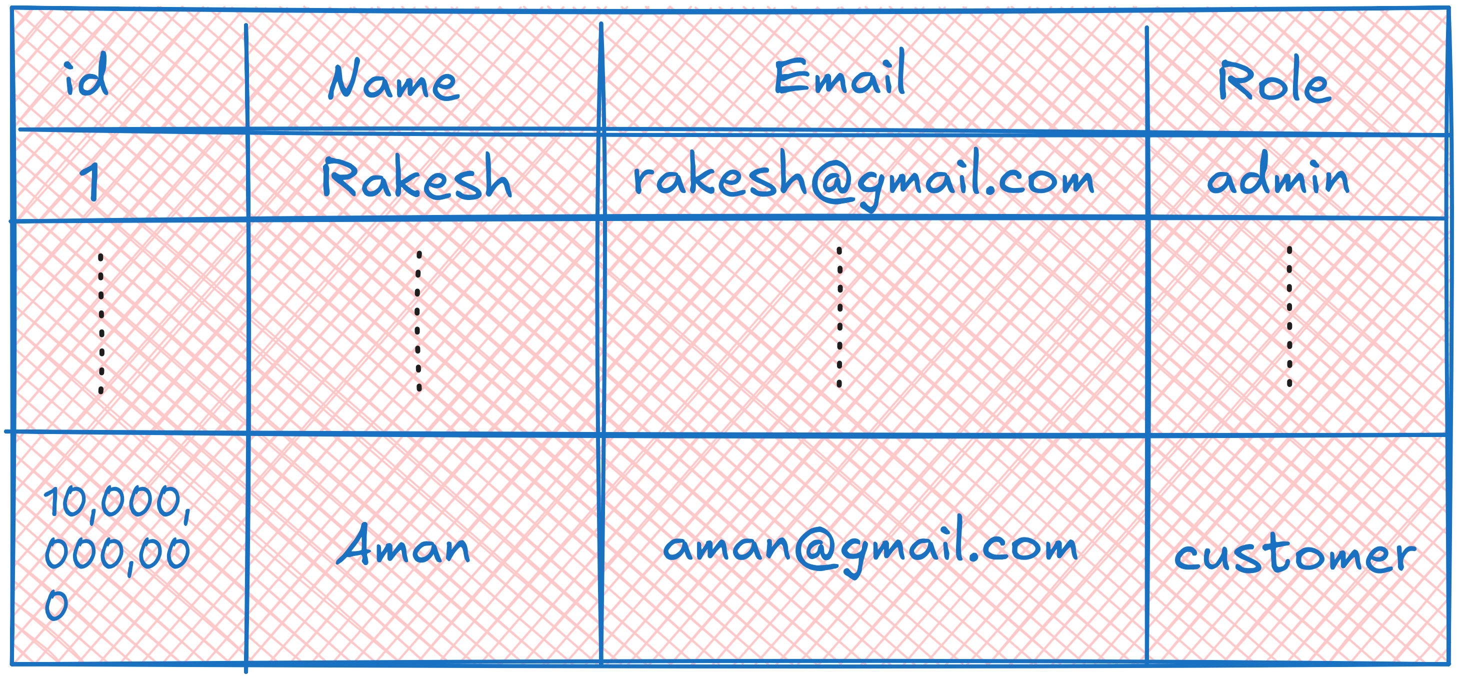
1. Without an index, the database will perform a sequential scan, which means it will go through each of the 10 billion rows one by one to find the result. This is incredibly time-consuming.

2. With an index (e.g., a B-tree index), while the query performance improves compared to a sequential scan, it can still be slow because the B-tree now has to manage 10 billion records. The large number of elements in the index introduces its own set of problems:

2.1 Height of the B-tree increases as the index grows. The more levels you need to traverse, the slower the lookups.

2.2 Memory and storage consumption of the index itself become significant.

2.3 Maintaining the index during frequent inserts, updates, and deletes incurs additional overhead.



id	Name	Email	Role
1	Rakesh	rakesh@gmail.com	admin
⋮	⋮	⋮	⋮
10,000,000,000	Aman	aman@gmail.com	customer

This is where solutions like Database Partitioning & Sharding comes into play.

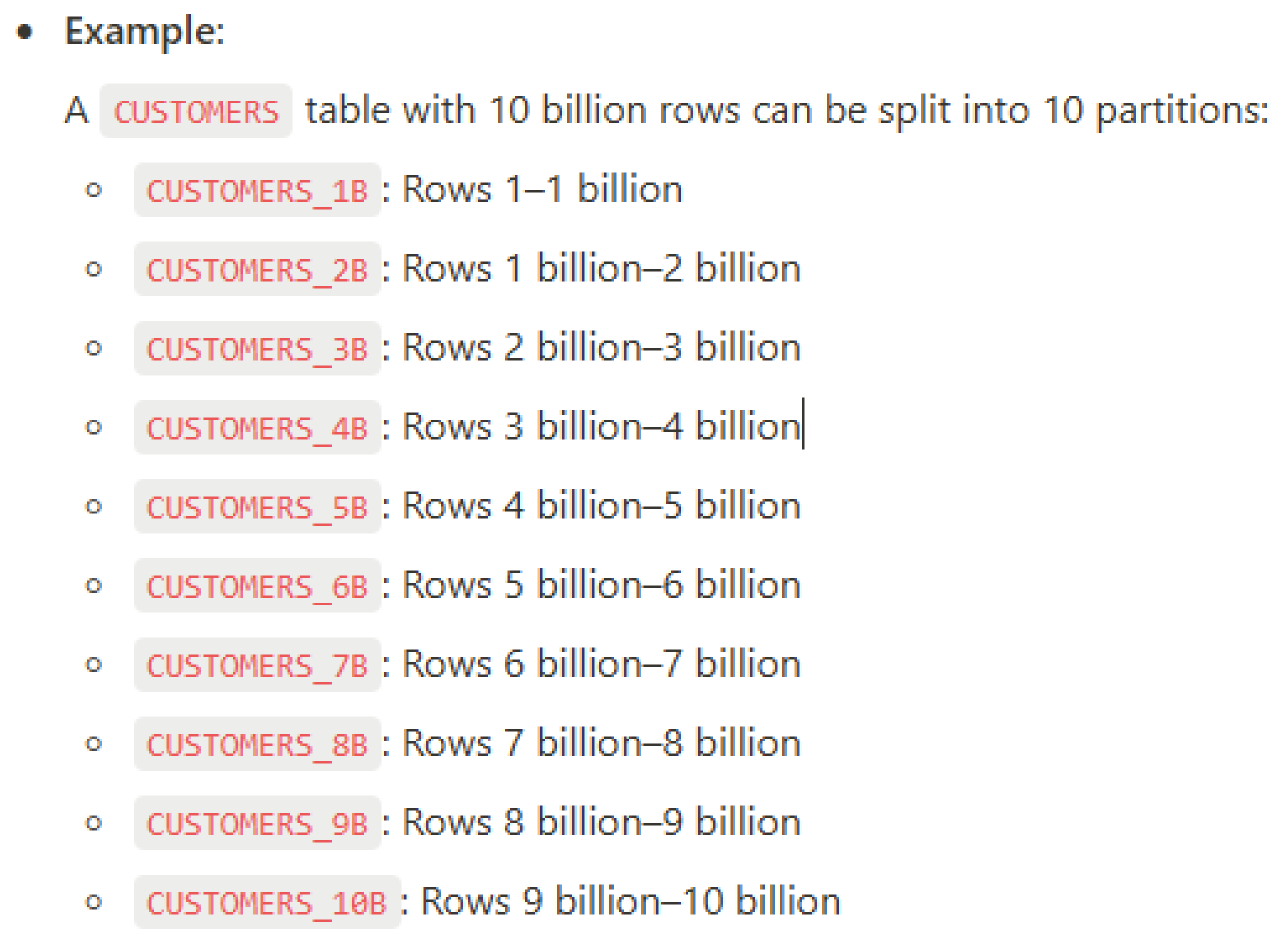
Database Partitioning

Database partitioning is a technique where we divide a large table into smaller tables, and the database decides which table (partition) to hit based on the partitioning scheme and the WHERE clause. We do not have to write any client logic for which partition to hit; the database handles it automatically. The main point here is that these partitions or smaller tables are stored on the same database server, and they are still logically connected to the main table.

Types of Database Partitioning

1. Horizontal Partitioning (most commonly used)

Horizontal partitioning splits rows of a table into different partitions. Each partition contains a subset of rows based on specific criteria.



- **Example:**

  A `CUSTOMERS` table with 10 billion rows can be split into 10 partitions:

  - `CUSTOMERS_1B`: Rows 1–1 billion
  - `CUSTOMERS_2B`: Rows 1 billion–2 billion
  - `CUSTOMERS_3B`: Rows 2 billion–3 billion
  - `CUSTOMERS_4B`: Rows 3 billion–4 billion
  - `CUSTOMERS_5B`: Rows 4 billion–5 billion
  - `CUSTOMERS_6B`: Rows 5 billion–6 billion
  - `CUSTOMERS_7B`: Rows 6 billion–7 billion
  - `CUSTOMERS_8B`: Rows 7 billion–8 billion
  - `CUSTOMERS_9B`: Rows 8 billion–9 billion
  - `CUSTOMERS_10B`: Rows 9 billion–10 billion

Partitioning Methods:

Horizontal partitioning splits rows of a table into different partitions. Each partition contains a subset of rows based on specific criteria.

1. By Range

Example: Partition by date ranges (e.g., January data in one partition, February in another).

2. By List

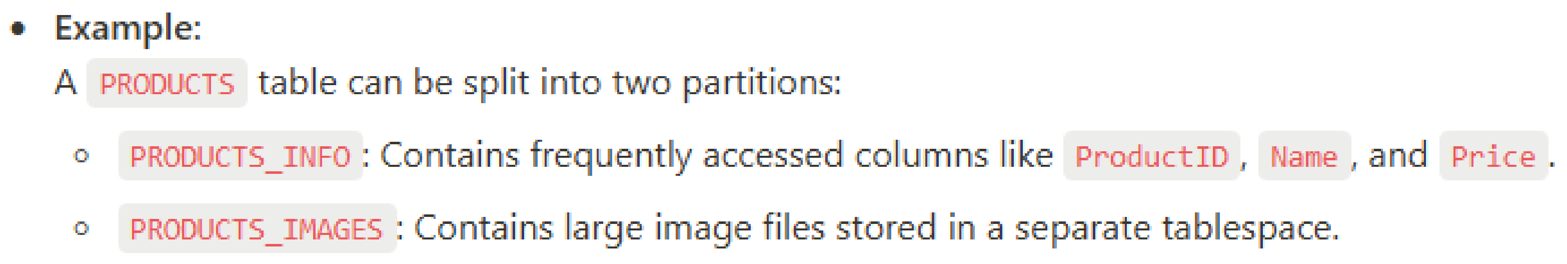
Example: Partition by discrete values like states (e.g., California customers in one partition, New York in another).

3. By Hash

Example: Use a hash function to distribute rows evenly across partitions. In case of UUID as primary key for a user table, we cannot use range-based partitioning effectively since UUIDs are random and not sequential, Instead, we can use **hash-based partitioning** to distribute rows evenly across partitions. 1. Use a hash function (e.g., MD5, SHA-256, or a built-in database hash function) to convert the UUID into a numeric value. 2. Divide the hash values into ranges or buckets corresponding to partitions. 3. Each row is assigned to a partition based on its hashed UUID value.

2. Vertical Partitioning

Vertical partitioning splits columns of a table into different partitions. This is useful when some columns are rarely accessed or are very large (e.g., BLOBs or images).



- Example:
  A `PRODUCTS` table can be split into two partitions:
  - `PRODUCTS_INFO`: Contains frequently accessed columns like `ProductID`, `Name`, and `Price`.
  - `PRODUCTS_IMAGES`: Contains large image files stored in a separate tablespace.

Advantages of Partitioning

1. Improved Query Performance:

Queries are faster because they only scan the relevant partition instead of the whole table.
Example: Searching for CustomerID = 700,001 will only scan the CUSTOMERS_800K partition.

2. Sequential Scans vs. Index Scans:

Partitioning enables sequential scans within a partition, which is faster than doing index scans over the entire table.

3. Efficient Bulk Loading:

New data can be added by creating a new partition, without affecting the existing data. For example, adding monthly sales records is simpler and faster.

4. Cost-Effective Storage:

Older or rarely accessed data (like logs from 5 years ago) can be moved to cheaper storage, while frequently accessed data stays on faster, more expensive drives.

Disadvantages of Partitioning

1. Row Movement Issues:

If a row needs to move from one partition to another (e.g., updating a customer's ID), it can be slow or may fail depending on the database system.

2. Inefficient Queries:

Poorly written queries might accidentally scan all partitions, resulting in slower performance than if the table were not partitioned.

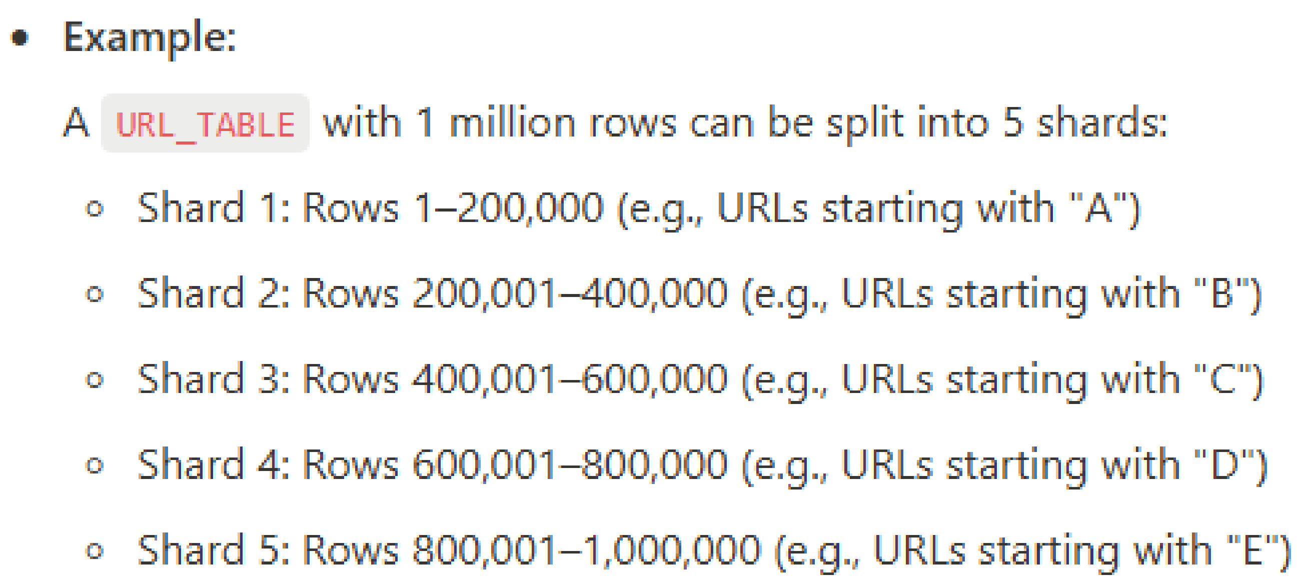
3. Schema Changes Are Complex:

Modifying the schema (e.g., adding a new column) can be challenging, especially if the database system does not handle it automatically.

Database Sharding

Database Sharding is a technique used to split a large database into smaller, independent pieces called shards. Each shard is stored on a separate database server, allowing for distributed storage and processing of data. This approach helps handle large datasets and high traffic loads.

Sharding splits a large database into smaller, independent pieces called shards , each stored on a separate server. Each shard holds a portion of the data, and together, all the shards form the complete dataset.



- **Example:**

  A `URL_TABLE` with 1 million rows can be split into 5 shards:

  - Shard 1: Rows 1–200,000 (e.g., URLs starting with "A")
  - Shard 2: Rows 200,001–400,000 (e.g., URLs starting with "B")
  - Shard 3: Rows 400,001–600,000 (e.g., URLs starting with "C")
  - Shard 4: Rows 600,001–800,000 (e.g., URLs starting with "D")
  - Shard 5: Rows 800,001–1,000,000 (e.g., URLs starting with "E")

Key Difference from Partitioning:

Unlike partitioning, where all partitions reside in the same database, shards are distributed across multiple servers.

Sharding Strategies

1.Range-Based Sharding

Data is divided into ranges based on values like IDs, dates, or numbers. For example, Shard 1 may contain URLs starting with "A", and Shard 2 may contain URLs starting with "B".

2. Hash-Based Sharding

A hash function is applied to a shard key (e.g., URLID) to determine which shard the data belongs to. For example, if the hash of URLID is 123456789 and the number of shards is 5, the shard number will be 123456789 % 5 = 4.

3. Directory-Based Sharding

A lookup table maps shard keys to specific shards. For example, a Shard_Directory table could map URL ranges to specific shards.

4. Geographical Sharding

Data is divided based on geographical regions. For example, Shard 1 may contain URLs accessed in North America, and Shard 2 may contain URLs accessed in Europe.

Advantages of Sharding

1. Improved Scalability:

Distributing data across multiple servers allows the system to handle larger datasets and higher traffic loads.

2. Faster Queries:

Queries are executed on smaller datasets, reducing latency and improving performance.

3. Fault Isolation:

If one shard fails, the others remain unaffected, improving fault tolerance.

4. Cost-Effective Scaling:

Adding new servers (shards) is often cheaper than upgrading a single server vertically.

Disadvantages of Sharding

1.Complexity in Querying:

Queries that span multiple shards require additional logic to aggregate results.

2 .Rebalancing Challenges:

Adding or removing shards can require redistributing data, which can be time-consuming and complex.

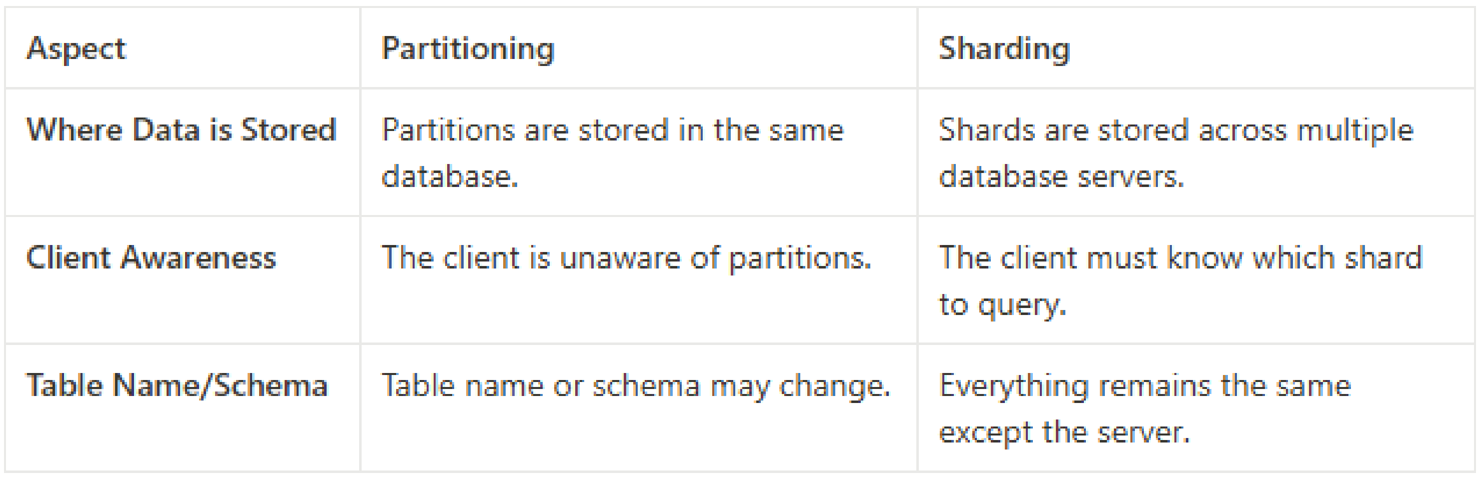
3. Data Hotspots:

Poorly chosen shard keys can lead to uneven distribution of data, causing some shards to become overloaded while others remain underutilized.

4. Increased Client Logic:

Clients must know which shard to query, adding complexity to application development.

Horizontal Partitioning vs Sharding



Aspect	Partitioning	Sharding
Where Data is Stored	Partitions are stored in the same database.	Shards are stored across multiple database servers.
Client Awareness	The client is unaware of partitions.	The client must know which shard to query.
Table Name/Schema	Table name or schema may change.	Everything remains the same except the server.