A Crash Course in Database Scaling Strategies

BYTEBYTEGO
JUL 04, 2024 · PAID

118 10 Share

Databases form the backbone of modern application development. They play a vital role in storing, managing, and retrieving data, enabling applications and services to function effectively.

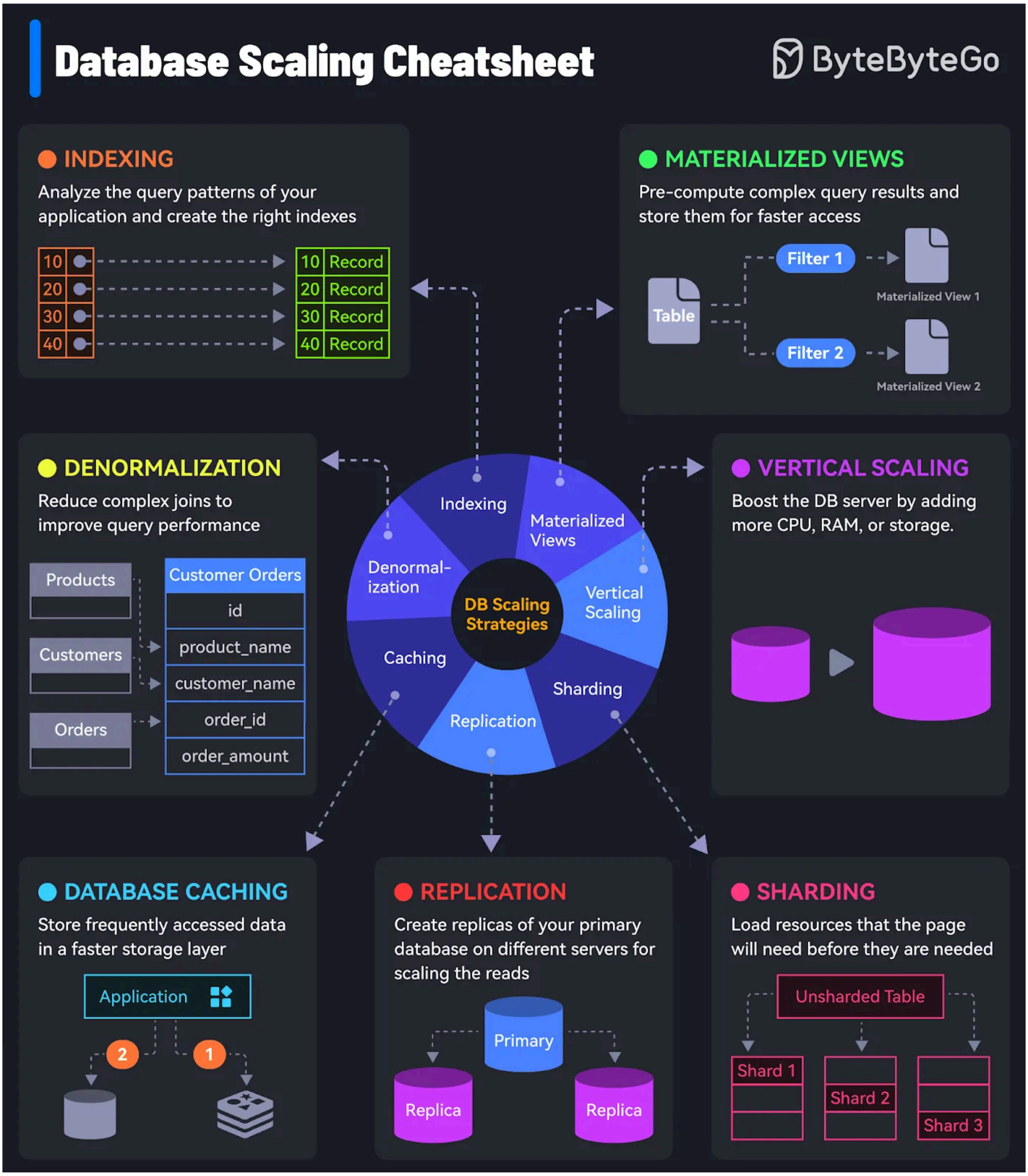
As applications gain popularity and attract a growing user base, databases face the challenge of handling ever-increasing data volumes, concurrent users, and complex queries.

It becomes critical to scale databases effectively to ensure optimal performance and a good user experience.

Database scaling is the process of adapting and expanding the database infrastructure to accommodate growth and maintain performance under increased load. It involves employing various techniques and strategies to distribute data efficiently, optimize query execution, and utilize hardware resources judiciously.

Organizations and developers must understand and implement the right database scaling strategy. Choosing the wrong strategies for a particular situation can result in more harm than good.

In this post, we will cover the most popular database scaling strategies in detail, discussing their benefits and trade-offs.



Database Scaling Cheatsheet

ByteByteGo

INDEXING

Analyze the query patterns of your application and create the right indexes

MATERIALIZED VIEWS

Pre-compute complex query results and store them for faster access

DENORMALIZATION

Reduce complex joins to improve query performance

VERTICAL SCALING

Boost the DB server by adding more CPU, RAM, or storage.

DATABASE CACHING

Store frequently accessed data in a faster storage layer

REPLICATION

Create replicas of your primary database on different servers for scaling the reads

SHARDING

Load resources that the page will need before they are needed

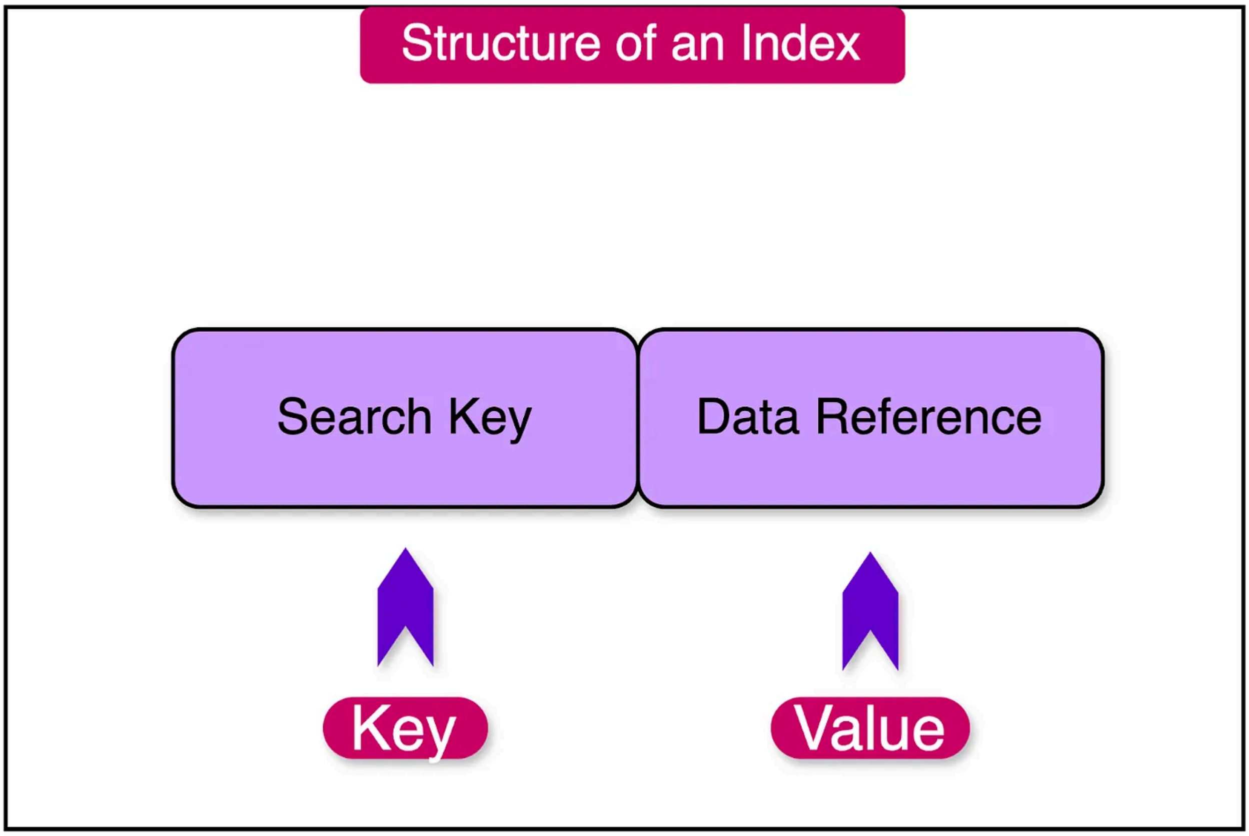
Indexing

Indexing is one of the foundational techniques to enhance the scalability and performance of databases.

An index can be thought of as a "table of contents" for a database. It contains references to the location of specific data within the main database tables, allowing for fast searching and retrieval.

By creating a separate data structure such as the index, databases can quickly locate and retrieve specific data without scanning through every single record in the main tables.

The index itself is a subset of the data, organized in a way that is optimized for efficient querying.

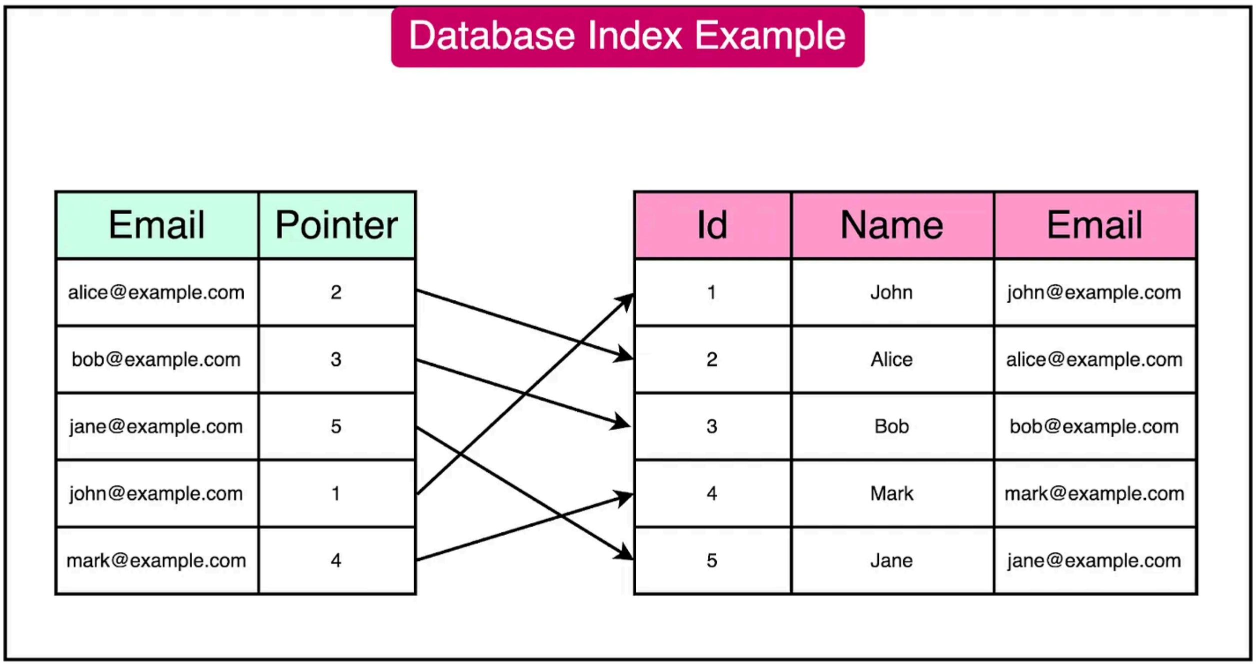


To understand the concept more clearly, let's consider a database table named "Customers" with columns such as "ID", "Name", "Email", and "City". If there is a frequent need to search for customers based on their email addresses, creating an index on the "Email" column can improve the search performance.

Without an index, searching for a customer by email would require the database to scan through every row in the "Customers" table until it finds the matching records. This process can be time-consuming, especially as the table grows in size, leading to slower query response times.

However, by creating an index on the "Email" column, the database can use the index data structure to quickly look up the desired email address and retrieve the corresponding row(s) directly. This removes the need for a full table scan, resulting in faster search operations.

See the diagram below for an example index on the “Email” column:



Email	Pointer
alice@example.com	2
bob@example.com	3
jane@example.com	5
john@example.com	1
mark@example.com	4

Id	Name	Email
1	John	john@example.com
2	Alice	alice@example.com
3	Bob	bob@example.com
4	Mark	mark@example.com
5	Jane	jane@example.com

Benefits of Indexing

Indexing offers several significant benefits that become increasingly important when dealing with large-scale databases:

- **Improved Query Performance:** As the data volume grows, the performance of queries can deteriorate if the database has to scan through a vast number of records. Creating the right indexes enables the database to quickly locate and retrieve specific subsets of data without the need for full table scans.
- **Reduced Resource Consumption:** Indexes allow the database to efficiently locate subsets of data, minimizing the amount of disk I/O and memory usage required.
- **Increased Concurrency:** With the help of indexes, the database can handle a higher volume of queries and accommodate more concurrent users. This improved concurrency is particularly beneficial in scenarios where the database has to scale to support a large number of users.

Trade-off with Indexing

It’s important to note that indexes come with multiple trade-offs.

- When an index is created on a column (such as the email address column from our example), the database stores the indexed data separately from the main table. This means that for each indexed column, there is an additional data structure that takes up space on the disk.
- Another trade-off to consider is the potential impact of indexes on write operations. When data is inserted, updated, or deleted in a table, the corresponding indexes should be updated to maintain accuracy and consistency. This extra step adds an overhead to write operations, which becomes more noticeable as the number of indexes on a table increases.

To strike the right balance, it’s important to carefully select the columns to index based on the specific query patterns and the application’s performance requirements.

Materialized Views

A materialized view is a database object that stores the result of a query as a separate, precomputed result set.

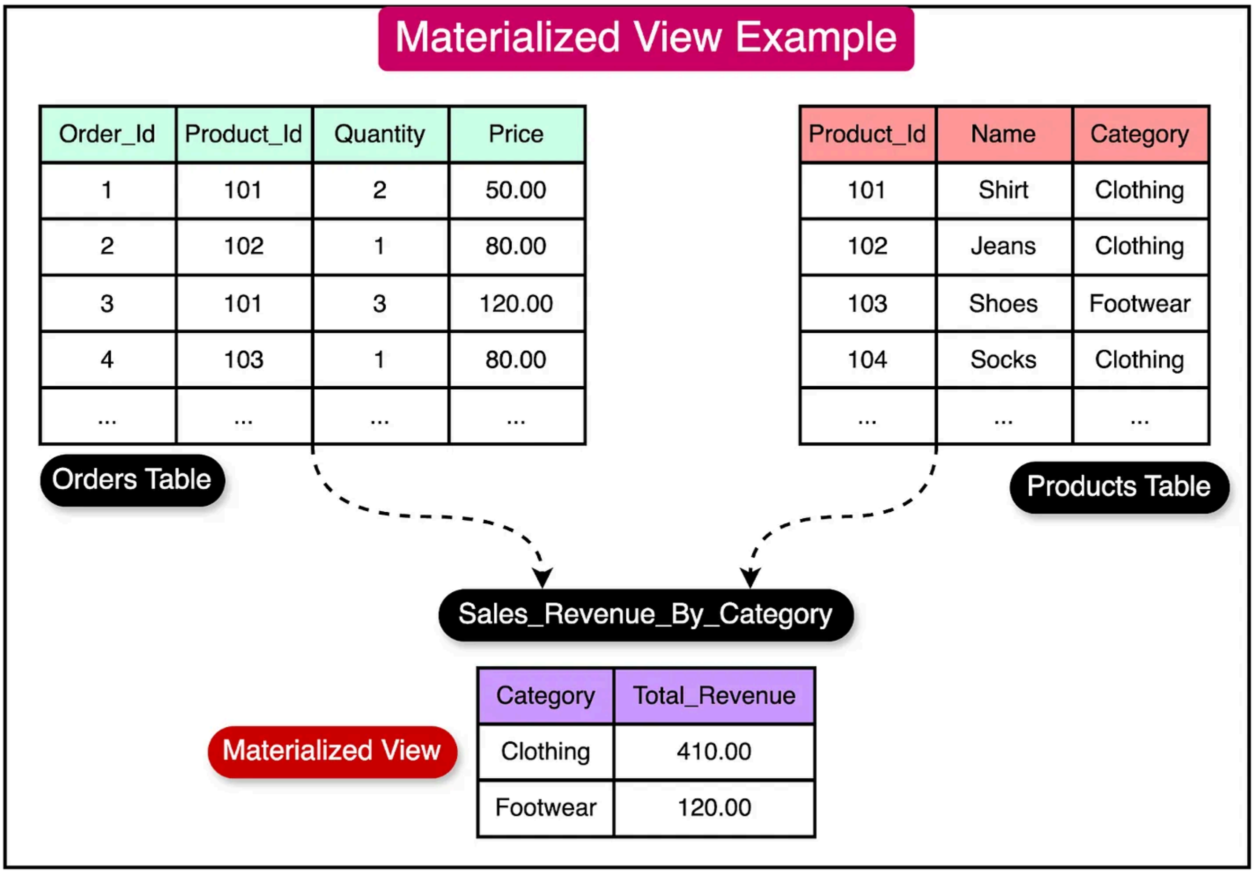
It is derived from one or more base tables or views and is maintained independently from the underlying data sources.

To explain the concept of materialized views, let's consider an e-commerce application with a large "Orders" table containing millions of records. The application frequently generates reports on the total sales revenue per product category.

Without materialized views, each report generation would require scanning the entire "Orders" table, joining it with the "Products" table to obtain category information, and performing aggregations to calculate the total revenue per category. As the data grows, this query becomes slower and resource-intensive.

By creating a materialized view that stores the pre-aggregated data, such as the total revenue per product category, the report generation process can be made faster.

See the diagram below that shows a materialized view storing the total revenue per product category.



Order_Id	Product_Id	Quantity	Price
1	101	2	50.00
2	102	1	80.00
3	101	3	120.00
4	103	1	80.00
...	...	...	...

Product_Id	Name	Category
101	Shirt	Clothing
102	Jeans	Clothing
103	Shoes	Footwear
104	Socks	Clothing
...	...	...

Category	Total_Revenue
Clothing	410.00
Footwear	120.00

The materialized view can be refreshed periodically, such as daily or hourly, to ensure the data remains up to date. Queries for the sales report can then be served directly from the materialized view, providing instant results without processing the entire "Orders" table.

Benefits of Materialized Views

Materialized views can greatly enhance database scalability in several ways:

- **Improved Query Performance:** Materialized views store pre-computed results, eliminating the need to execute complex and time-consuming queries repeatedly.
- **Reduced Load on Base Tables:** By storing computationally expensive query results in materialized views, the load on the base tables is reduced.

Trade-offs with Materialized Views

While materialized views offer a significant scalability boost, there are some trade-offs to keep in mind:

- Materialized views consume additional storage space since they store a separate copy of the result set.
- Refreshing materialized views can be time-consuming, especially for large datasets.
- Materialized views are eventually consistent with the source data. In other words, the materialized view can contain stale data for a brief period.

Denormalization

In a normalized database design, data is organized into separate tables to minimize redundancy and ensure data integrity. Each table represents a single entity or concept and table relationships are established using foreign keys. This approach follows the principles of normalization, which aim to reduce data duplication and maintain data consistency.

However, strict adherence to normalization rules can sometimes lead to performance challenges, particularly when dealing with complex queries that involve multiple joins across tables.

This is where denormalization comes into play.

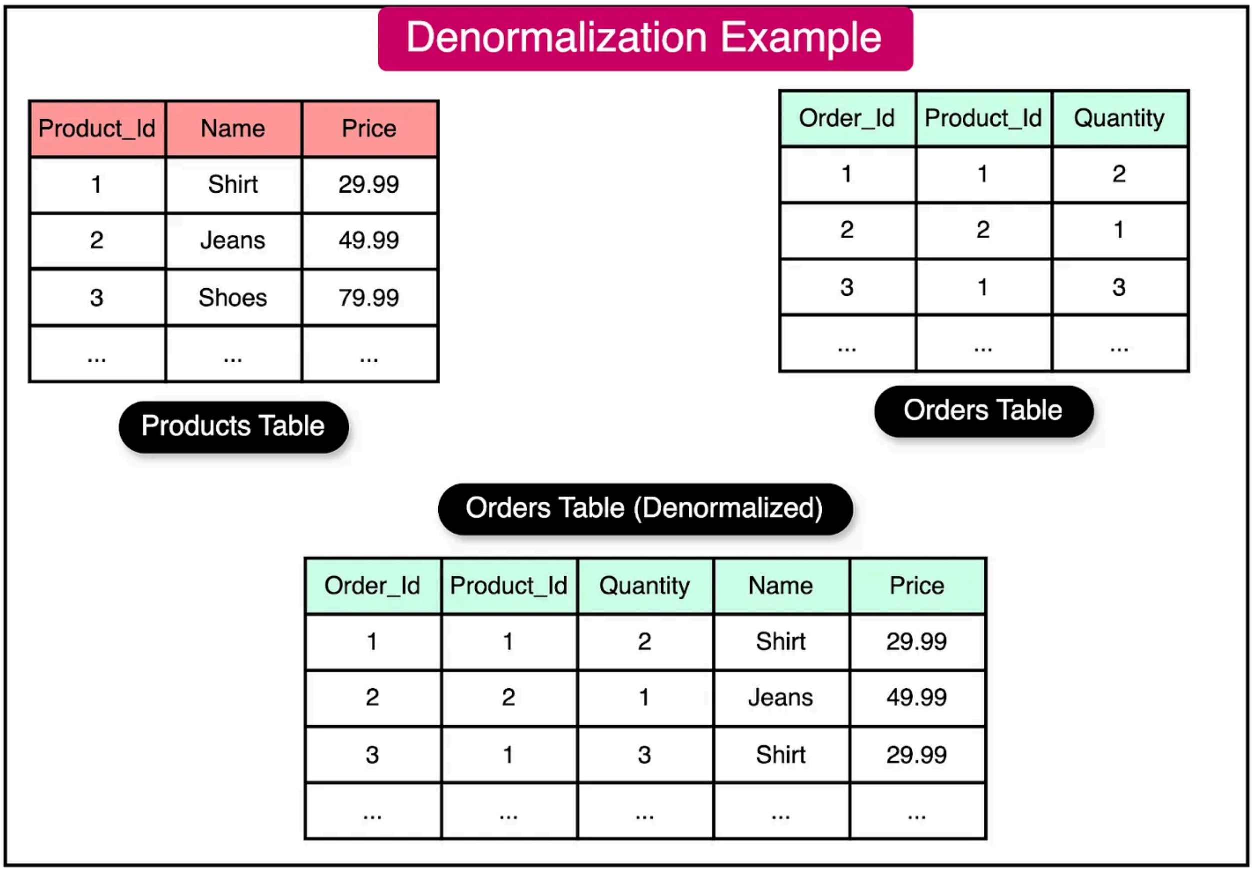
Denormalization is a technique that relaxes the strict normalization rules and allows for controlled data redundancy. It involves strategically duplicating data across multiple tables to optimize query performance. The goal is to reduce the number of joins and computations required to retrieve data, thereby improving query speed and scalability.

To understand the concept of denormalization, consider an e-commerce application with a "Products" table and an "Orders" table.

In a normalized design, the "Orders" table would store only the foreign key reference to the "Products" table. A join between the two tables would be necessary to retrieve the product details with the order information.

However, as the number of orders grows, the join operation can become a performance bottleneck if the application frequently needs to display the product name and price alongside the order details. In such cases, denormalization can be applied to improve query performance.

The diagram below shows an example of applying denormalization on the "Orders" table:



Denormalization Example

Product_Id	Name	Price
1	Shirt	29.99
2	Jeans	49.99
3	Shoes	79.99
...	...	...

Products Table

Order_Id	Product_Id	Quantity
1	1	2
2	2	1
3	1	3
...	...	...

Orders Table

Orders Table (Denormalized)

Order_Id	Product_Id	Quantity	Name	Price
1	1	2	Shirt	29.99
2	2	1	Jeans	49.99
3	1	3	Shirt	29.99
...	...	...	...	...

By denormalizing the database and storing the product name and price directly in the "Orders" table, the query to retrieve order details along with product information becomes simpler and faster. The redundant data eliminates the need for the join, allowing the database to scale better under high query loads.

How Denormalization Helps With Scalability?

Denormalization can contribute to database scalability in several ways:

- **Faster Query Execution:** Eliminating or reducing joins can significantly speed up query execution, especially for frequently accessed or performance-critical queries. In other words, the database can handle a higher volume of concurrent queries, improving overall scalability.

- **Reduced Data Retrieval Overhead**: Since derived data is stored alongside the main data in the same table, there is less need to perform expensive on-the-fly calculations.
- **Improved Read Performance:** Denormalization is particularly beneficial for scaling read-heavy workloads since queries can access information without the need to join multiple tables.

Trade-offs with Denormalization

While denormalization can improve query performance and scalability, it comes with some trade-offs:

- Denormalization introduces data redundancy, which can increase storage requirements.
- Denormalization makes data modification operations (inserts, updates, deletes) more complex and slower, as the redundant data needs to be kept in sync across multiple tables.
- Denormalization may compromise data consistency if not implemented correctly.

Vertical Scaling

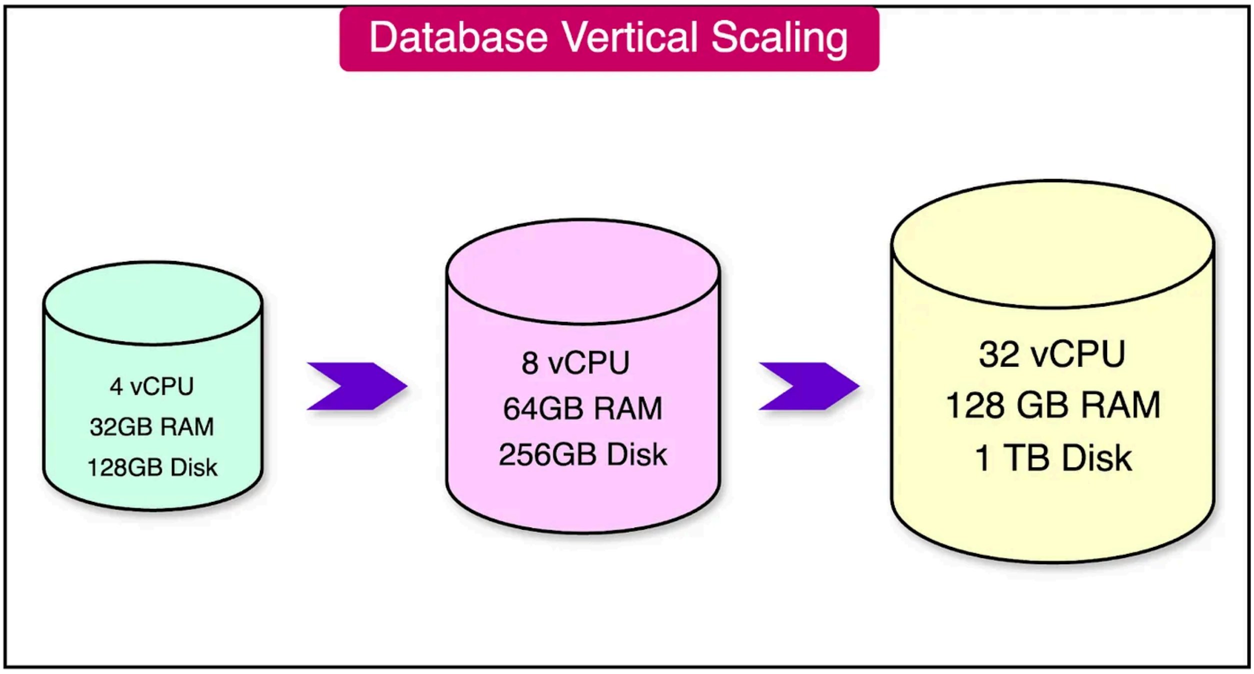
Vertical scaling, also known as "scaling up," is a technique to improve database performance and scalability by increasing the hardware resources of a single server.

This approach focuses on enhancing the capabilities of an individual server by allocating more resources to it.

The process of vertical scaling can involve several key upgrades:

- Replacing the existing CPU with a faster or multi-core processor to increase processing power and enable faster query execution.
- Adding more RAM to the server to increase memory capacity, allowing for improved caching.

- Upgrading to faster storage devices, such as solid-state drives (SSDs).



To understand the benefits of vertical scaling, let's consider an e-commerce application that experiences a surge in traffic during peak shopping seasons. As the load on the database server increases, query response times may suffer, leading to a poor user experience.

To address this scalability challenge, the application owner decides to vertically scale the database server. They upgrade the server with a more powerful CPU, double the RAM capacity, and replace the hard disk drives (HDDs) with SSDs.

These hardware enhancements significantly improve the database server's performance. The faster CPU and increased memory enable quicker query execution and efficient caching, while the SSDs provide faster data retrieval and write speeds.

As a result, the database server can handle more concurrent users and deliver an improved performance.

Benefits of Vertical Scaling with Database Scalability

Vertical scaling can contribute to database scalability in several ways:

- **Improved Query Performance:** With more powerful hardware, the database can execute queries faster. Additional RAM allows for a larger memory cache, enabling faster data retrieval.

- **Increased Concurrency:** Vertical scaling allows the database server to handle a higher number of concurrent users and connections.
- **Larger Dataset Handling:** Vertical scaling enables the database server to accommodate larger datasets.

Trade-offs with Vertical Scaling

Some trade-offs to keep in mind while using vertical scaling are as follows:

- Vertical scaling has limits based on the maximum hardware capabilities available. There is a ceiling to how much a single server can be upgraded.
- Vertical scaling does not provide the same level of fault tolerance and high availability as horizontal scaling. If the single database server fails, the entire database becomes unavailable.
- Vertical scaling may require downtime during the hardware upgrade process, impacting the availability of the database.

Caching

Caching is a technique that involves storing frequently accessed data in a high-speed storage layer, separate from the primary database.

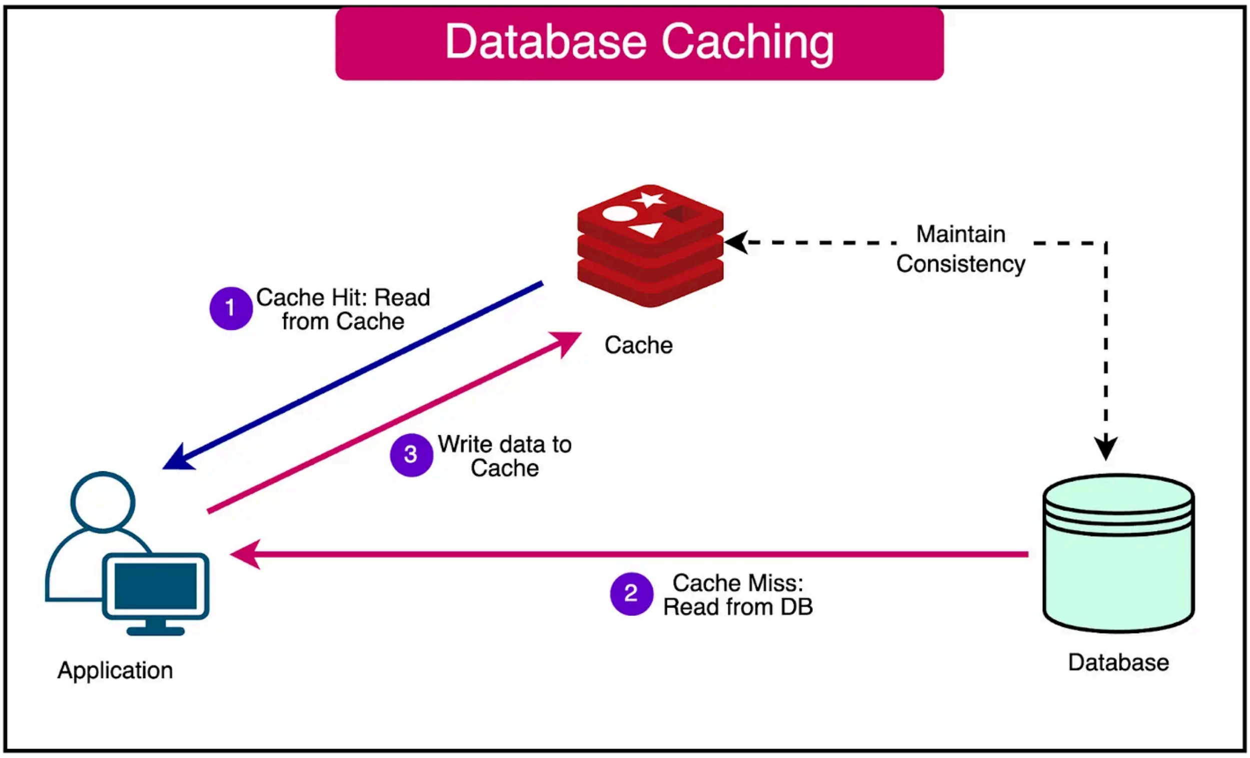
This high-speed storage layer is typically implemented using memory or fast disk storage. The primary goal of caching is to reduce the number of requests made to the database by serving frequently accessed data directly from the cache.

The cache works as follows:

- When an application receives a request for data, it first checks the cache to determine if the requested data is available.
- If the data is found in the cache, known as a cache hit, it is quickly retrieved from the cache without the need to query the database. This significantly reduces the response time and improves the application’s performance.
- On the other hand, if the requested data is not found in the cache, referred to as a cache miss, the application retrieves the data from the primary database.

- In addition to returning the data to the client, the application stores a copy of the retrieved data in the cache. This ensures that subsequent requests for the same data can be served from the cache, removing the need for repeated database queries.

The diagram below shows the caching process with a database:



As an example, consider a social media application displaying user profiles and their recent activities. When a user visits a profile page, the application needs to retrieve the user's information and their latest posts from the database.

Without caching, every time a user profile is viewed, the application would query the database to fetch the user's data and their recent activities. As the number of users and profile views increases, the database can become overwhelmed with requests, leading to slower response times and reduced scalability.

The application can use a cache to store the frequently accessed or popular user profiles and their recent activities. This approach reduces the number of queries made to the database, improves response times, and allows the application to handle a higher volume of profile views without overloading the database.

How does Caching help with Scalability?

Caching improves the scalability of a database in several ways:

- **Reduced Database Load:** Caching reduces the number of queries made to the database, allowing it to handle more concurrent requests and scale out.
- **Faster Query Response:** Caching eliminates the need for disk I/O, thereby reducing query response times.
- **Better Read Performance:** Caching helps in distributing the read workload. By serving a significant portion of the read requests from the cache, the database can focus on handling write operations and more complex queries at scale.

Trade-offs with Caching

Some trade-offs to consider while caching are as follows:

- Caching introduces additional complexity to the system architecture.
- Caching consumes additional memory or disk space to store the cached data. This incurs extra cost.
- Stale data in the cache can lead to cache inconsistency. Proper invalidation strategies have to be adopted to keep the cache consistent with the database.
- Caching may not be effective for rarely accessed or frequently updated data, as the cache hit ratio would be low.

Horizontal Scaling

Horizontal scaling, known as "scaling out," involves adding more servers or nodes to a database system to distribute the workload and improve performance.

It provides better scalability and fault tolerance compared to vertical scaling by eliminating the single point of failure.

There are two main database scaling strategies when it comes to horizontal scaling:

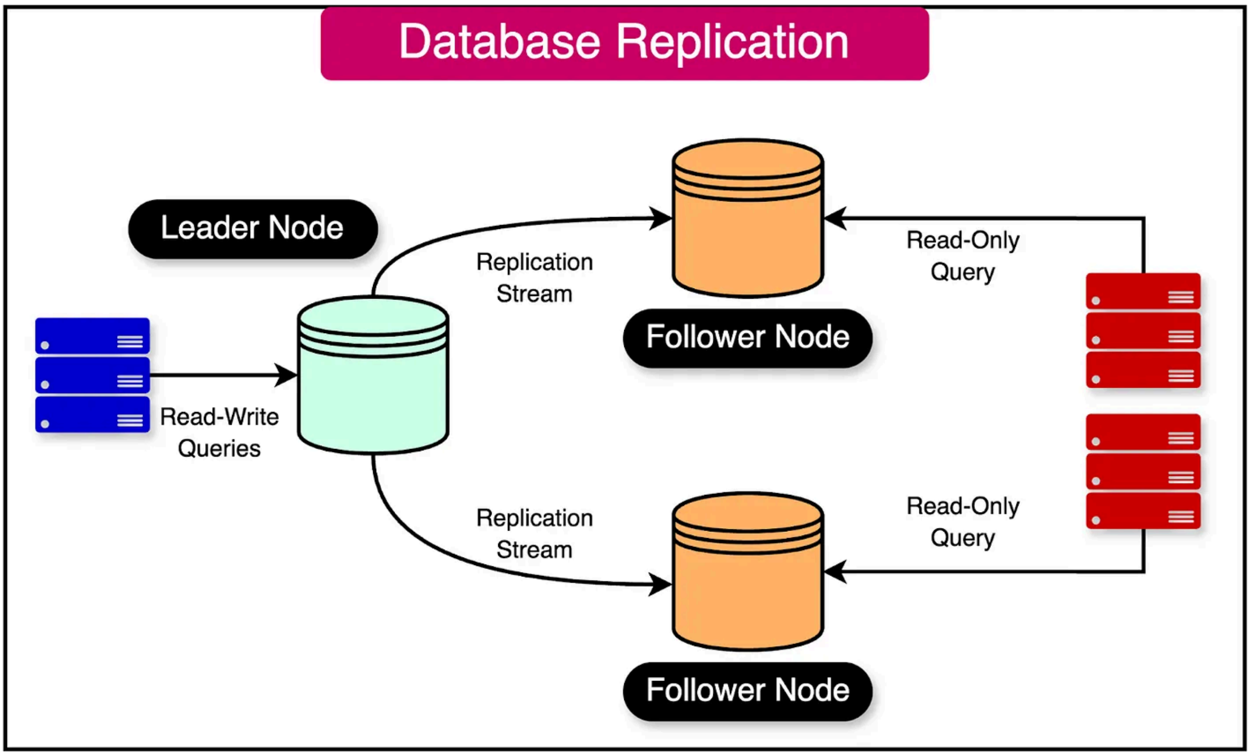
Replication

Replication is a technique employed in database systems to create and maintain multiple copies of data across different servers or nodes. By distributing data across multiple replicas, replication enhances data availability, fault tolerance, and scalability.

In a typical leader-follower replication model, one node is designated as the leader while the other nodes are referred to as followers. The leader node handles all write operations, such as inserts, updates, and deletes ensuring data consistency and integrity. It can also handle a few important read operations that need read-after-write consistency.

Whenever a write operation is performed on the leader node, the changes are automatically replicated to the follower nodes. This replication process ensures that the follower nodes maintain an exact copy of the data stored on the leader node.

See the diagram below that shows a typical leader-follower replication setup:



The follower nodes, on the other hand, are used to handle read operations. They can serve read queries in parallel with the leader node, effectively distributing the read workload across multiple nodes. This distribution of read operations helps in scaling the database horizontally and improving its performance.

Benefits of Replication

Replication offers several benefits for database scalability:

- **Improved Read Performance**: By distributing the read workload across multiple replica servers, replication allows for parallel processing of read queries. This reduces the load on the leader node and improves the overall read performance of the workload.
- **High Availability**: Replication provides high availability by maintaining multiple copies of the data. If the leader node fails, one of the follower nodes can be promoted to take over the leader's responsibilities.
- **Durability:** Replication improves the durability of the database by creating multiple copies of the data.

Trade-offs with Replication

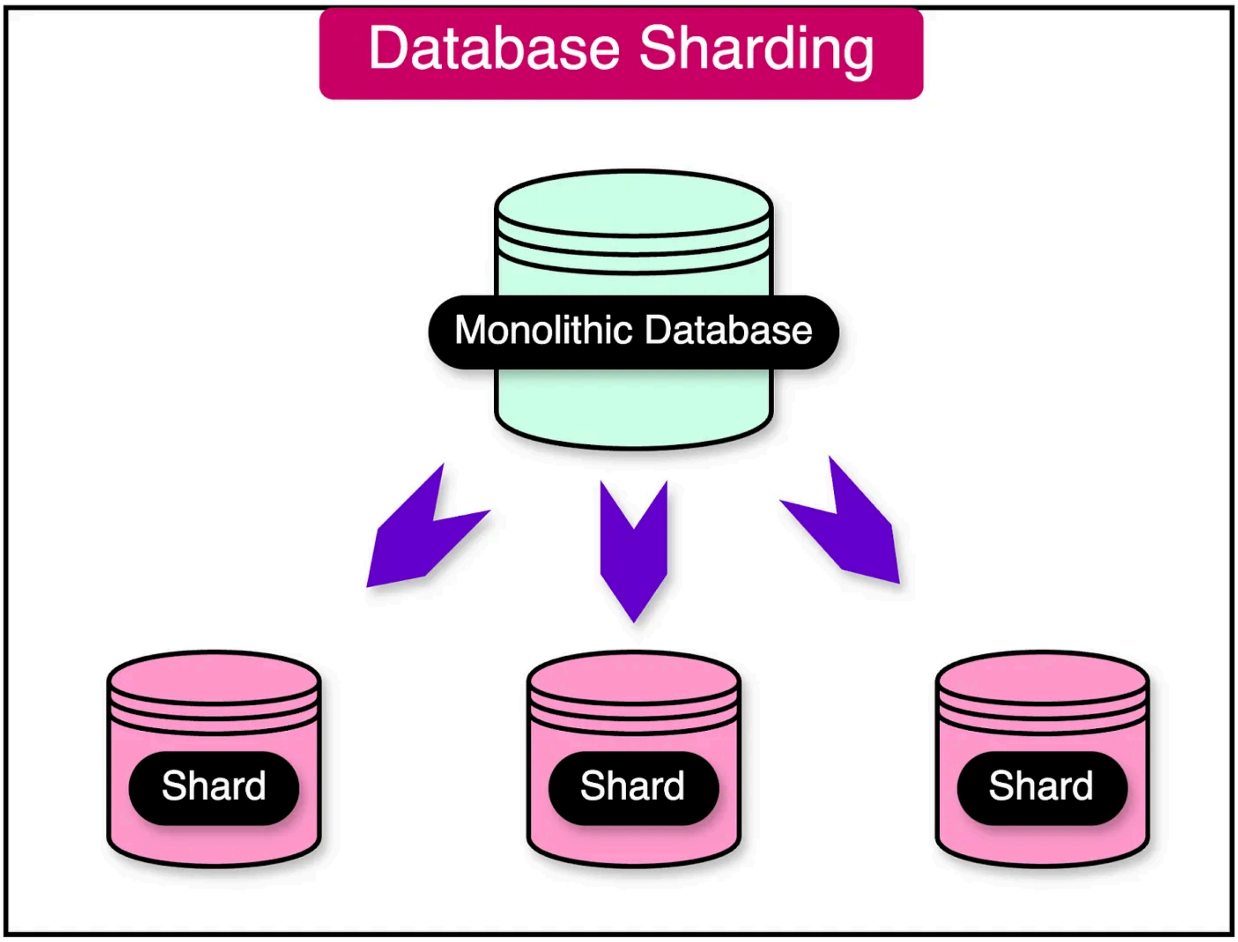
Some trade-offs to consider with replication are as follows:

- Replication introduces some latency in data synchronization between the primary and replica servers. There will be a delay (known as replication lag) before changes made on the leader are reflected in the followers.
- Replication adds complexity to the database setup.
- In leader-follower replication, write operations are still handled by a single server, which can become a bottleneck for write-heavy workloads. This might require the adoption of other replication types like multi-leader replication.

Sharding

Database sharding is a technique to partition a single large database into smaller, more manageable units called shards. By dividing the data into independent pieces, sharding enables databases to scale horizontally and handle increased data volume.

In a sharded database architecture, the data is distributed across multiple shards based on a specific sharding key. The choice of the sharding key is crucial as it determines how the data is allocated to different shards.



Common sharding strategies are as follows:

- **Range-based Sharding:** Data is partitioned based on a range of values of the sharding key.
- **Hash-based Sharding:** It involves applying a hash function to the sharding key to determine the target shard for each data record.
- **Directory-based Sharding:** A separate lookup table is maintained to map the sharding key to the corresponding shard.

Each shard in a sharded database architecture operates independently, hosting a subset of the overall data. Queries and write operations are routed to the appropriate shard based on the sharding key.

Benefits of Sharding:

Sharding offers several benefits for database scalability:

- **Horizontal Scalability:** Sharding allows for horizontal scaling of the database by adding more shards as needed.
- **Improved Performance:** Queries and write operations can be processed in parallel across the shards, improving overall performance.
- **Reduced Hardware Costs:** Sharding allows for the use of commodity hardware instead of expensive high-end machines for scalability.

Trade-offs with Sharding

With sharding, there are some trade-offs to keep in mind:

- Sharding introduces complexity to the database architecture and application design, especially when handling cross-shard queries and transactions.
- Resharding, or rebalancing data across shards, can be a complex and time-consuming process as the data volume grows.
- Joining data across shards can be challenging and may require additional effort or denormalization techniques.

Summary

In this article, we explored essential strategies for scaling databases to accommodate increasing data volume, traffic, and performance demands.

These strategies provide various approaches to optimize database performance, distribute workload, and facilitate growth.

Let's summarize each of these strategies:

- **Indexing:** Enhances query performance and scalability by creating separate data structures that enable faster data retrieval without scanning the entire table.
- **Materialized Views:** Materialized views store precomputed query results, allowing for faster access to frequently requested data.
- **Denormalization:** Denormalization involves intentionally adding redundant data to tables to minimize the need for complex joins and speed up queries.
- **Vertical Scaling:** Focuses on improving performance by increasing the hardware resources such as CPU, RAM, and storage, of a single database server.

- **Caching:** Caching involves storing frequently accessed data in a fast-access memory layer, separate from the primary database.
- **Replication:** Creates multiple copies of the database across different servers, enabling read scalability and high availability.
- **Sharding:** Technique to partition the database into smaller, independent shards, each storing a subset of the overall data.

In the end, an important point to understand is that there is no need to use all strategies together. Application developers should choose the appropriate strategies based on workload requirements, team capability, and cost-related concerns.

118 Likes · 10 Restacks

Comments

Write a comment...