**Chapter 2. Storage Types and Relation Stores**

**A NOTE FOR EARLY RELEASE READERS**

With Early Release ebooks, you get books in their earliest form—the authors’ raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *mpotter@oreilly.com*.

In modern computing systems, data storage is an essential component that plays a vital role in system design. As businesses and organizations continue to scale and grow, they generate and store vast amounts of data, which makes it important to have a scalable and reliable storage infrastructure that can keep up with the increasing demands. Efficient and reliable storage of data is necessary to ensure that applications and services can function effectively and provide optimal performance to end-users. There are various types of data storage solutions available, ranging from traditional file-based systems to more modern block and object stores. Additionally, databases play a significant role in storing and managing structured data, making them a critical aspect of system design.

In this chapter, we will explore the different types of data storage solutions available in the context of system design. We will start by discussing the traditional file-based storage systems and their limitations, before moving on to block and object stores, which have become increasingly popular in recent years due to their ability to handle large amounts of unstructured data. We will also explore the different types of databases available in the relational world in this chapter followed by the non-relational stores in the next chapter, as well as the advantages and disadvantages of each.

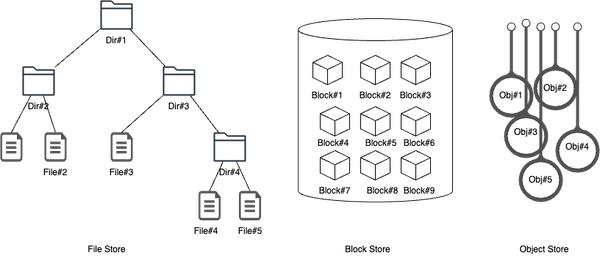
As we examine the challenges of scaling relational databases, we’ll explore various techniques such as partitioning, indexes, replication, federation, sharding, and denormalization that can be used in relational stores to improve performance and meet the demands of large-scale applications.

By the end of this chapter, you will have developed a solid understanding of the various data storage solutions available, the strengths and weaknesses of different relational database types and how they can be used to design systems that are efficient, reliable, and scalable. This chapter will serve the foundation of the concepts to compare and choose between different AWS relational database offerings including AWS RDS flavors and AWS Aurora covered in Chapter 10.

**Data Storage Format**

The evolution of data storage hardware over the years posed a challenging problem—how to store the data in a particular format, which hides out the underlying storage hardware and can be read, written and modified across different hardwares and their evolutions. This lead the computer scientists and engineers to come up with device drivers, particularly storage drivers, which were installed on different operating systems to allow working with the data on that particular storage hardware, stored in a particular “format”, agnostic to underlying hardware.

Three particular formats as shown in [Figure 2-1](https://learning.oreilly.com/library/view/system-design-on/9781098146887/ch02.html#fig_1_storage_abstraction_in_file_store_block_store_and),  emerged based on how the data is logically arranged on the storage hardware: file-based, block-based, and object-based storage, each with their own capabilities and limitations. File storage organizes and represents the data as a hierarchy of files and folders; Block storage chunks data into arbitrarily organized fixed-size blocks; and Object store organizes them as whole objects linked with the associated metadata. Let’s go through each of these storage formats in detail.



**Figure 2-1. Storage abstraction in File store, Block store and Object store**

**File Storage**

File storage refers to the storage of data in a folder, just like how paper documents are kept in a filing cabinet. When the data is needed, the computer must know the path to find it, which can even be a long arduous path string.

Data in files is organized and retrieved using metadata, like a library card catalog. Consider a closet full of file cabinets. Every document is arranged in some form of logical hierarchy - by cabinet, by drawer, by folder, by file and then, by paper. This is where the concept of hierarchical storage comes from and the file system provides similar hierarchical structure to organize files, and data is stored in blocks or pages on disk.

It is the oldest and most widely used data storage system for direct and network-attached systems, great for storing complex files and easier to navigate compared to other formats due to the logical hierarchy. Any time you access files on your personal computer, you leverage the file storage.

The caveat with file storage is, just like with your filing cabinet, that the virtual drawer can only open so far. File-based storage systems must scale out by adding more systems, rather than scale up by adding more capacity.

File-based storage is commonly used for storing structured data, such as documents, images, videos, and audio files.

**NOTE**

AWS Elastic File Store (EFS) is a scalable, fully managed file storage service offering by AWS that provides shared file storage for EC2 instances, enabling multiple instances to access the same data concurrently, making it suitable for applications requiring shared access to files. We will cover it in more detail in Chapter 10 - AWS Storage Services.

**Block Storage**

Block-based storage organizes data into fixed-size blocks or pages, which are stored on disk or flash memory. Each block is assigned a unique address, and data can be read or written to individual blocks. Block storage is designed to separate the data from the user’s environment and distribute it across multiple environments that are better suited to serve the data. This means, some block data can be on a Windows environment, some on a Linux environment, and so on. When data is requested, the storage software reassembles the data blocks from these environments and delivers them back to the user. Typically used in storage-area network (SAN) environments, block storage requires a functioning server to operate.

Since, unlike file storage, block storage doesn’t rely on a single path to data, it can be retrieved quickly. Each block lives on its own and can be partitioned so it can be accessed in a different operating system, which gives the user complete freedom to configure their data. It’s an efficient and reliable way to store data and is easy to use and manage. It works well with enterprises performing big transactions and those that deploy huge databases, meaning the more data you need to store, the better off you’ll be with block storage.

The caveat with Block storage is that it can be expensive. It has limited capability to handle metadata, which means it needs to be dealt with at the application or database level, adding another complexity for a developer or systems administrator to worry about.

Block-based storage is commonly used in enterprise storage systems, such as SAN and NAS, and it provides better performance, reliability, and scalability than file-based storage.

**NOTE**

AWS Elastic Block Storage (EBS) offers scalable block storage volumes on AWS cloud that can be attached to EC2 instances, providing durable, high-performance storage for applications that require low-latency access to data, such as databases and applications that need block-level storage. We will cover it in more detail in Chapter 10 - AWS Storage Services.

**Object Storage**

Object-based storage, or object storage, is a type of storage architecture where data is broken down into discrete units called objects and kept in a single repository, rather than in traditional file and folder structures or server blocks. These objects are stored across distributed hardware and are accessed using a unique identifier and metadata that describes the data, including information like age, security and access contingencies, and even details like the location equipment used to create a video file.

Object storage volumes function as modular units, with each one being a self-contained repository of data. Retrieval of this data is done using the unique identifier and metadata, which distributes the workload and enables administrators to apply policies for more efficient searches. Object storage is also known for its scalability, cost efficiency, and suitability for static data and unstructured data. It has a simple HTTP API that is used by most clients in all programming languages.

However, there are limitations to object storage. Objects cannot be modified, meaning that data must be written entirely at once. Also, object storage does not work well with traditional databases because writing objects is a slow process and programming an app to use object storage API is not as straightforward as using file storage.

**NOTE**

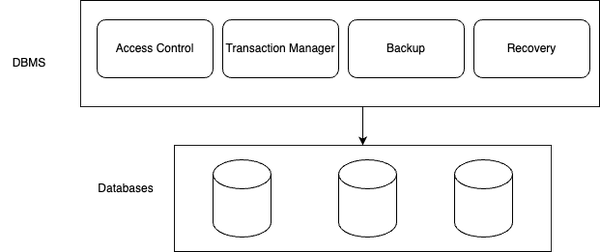
AWS S3 (Simple Storage Service) is an object storage service offering by AWS that provides highly scalable and durable storage for a wide range of data types, accessible via APIs. It is suitable for storing and retrieving large amounts of data, backups, static website content, and as a data lake for analytics. We will cover it in more detail in Chapter 10 - AWS Storage Services.

In summary, the choice of data storage format depends on the type of data being stored, as well as the performance, reliability, and scalability requirements of the system. File-based storage is best suited for structured data, while block-based and object-based storage are more appropriate for unstructured data. Block-based storage provides better performance and reliability than file-based storage, while object-based storage offers better scalability and cost-effectiveness.

We covered different data storage formats and their comparison in terms of usage, cost, performance and scalability. Next let’s discuss how data is stored and structured in databases for easy retrieval and processing.

A database is a structured collection of data that is organized and stored in a computer system for easy access, retrieval, and management. It is designed to efficiently store, retrieve, and manipulate large volumes of data. A database typically consists of one or more tables, which are organized into rows and columns. Each row represents a unique record in the database, while each column represents a specific attribute or piece of information about the record.

Database management systems (DBMS), on the other hand as show in [Figure 2-2](https://learning.oreilly.com/library/view/system-design-on/9781098146887/ch02.html#fig_2_databases_vs_database_management_system) are a collection of software systems that sit on top of a database, acting as a bridge between the database and users. DBMS offers multiple interfaces or APIs that enable users to store, retrieve, and manipulate data. Additionally, DBMS provides a range of features related to transactions, recovery, backups, concurrency, authentication and authorization, metadata catalog, and other capabilities.



**Figure 2-2. Databases vs Database Management System**

Databases can be classified into two major types, based on their structure, usage, and functionality: relational and non-relational databases. Let’s go through relational databases in detail in this chapter. We’ll be covering non-relational databases in Chapter 3.

**Relational Databases**

Relational databases use a set of tables with relationships between them to organize data.

Relational databases are the most commonly used type of database, designed to store and manage large amounts of structured data. They are based on the [relational model](https://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf), which was first introduced by Edgar F. Codd in the 1970s. This model organizes data into tables, which can be related to each other using keys.

Tables in a relational database are organized into rows and columns, similar to a spreadsheet. Each row represents a record, and each column represents an attribute or a piece of data about the record. The columns are defined by a data type, which specifies the type of data that can be stored in that column.

[Figure 2-3](https://learning.oreilly.com/library/view/system-design-on/9781098146887/ch02.html#fig_3_logical_schema_design_of_order_database) below shows a sample implementation of logical schema design of an Order database.



**Figure 2-3. Logical schema design of Order Database**

The list below discusses these concepts, which form the logical components of a database schema design, in detail.

*Tables*

Tables are the fundamental units of a relational database. They are used to store and organize data in rows and columns. Each table represents a specific entity or concept, and each row in the table represents an individual instance of that entity, while each column represents a specific attribute or property.

*Rows*

Rows, also known as records or tuples, represent individual instances of data stored in a table. Each row contains data values that correspond to the attributes defined by the table’s columns. Rows are unique within a table, typically identified by a primary key.

*Columns*

Columns, also known as fields or attributes, represent the specific properties or characteristics of the data stored in a table. Each column has a defined data type that determines the kind of data it can store, such as integers, strings, dates, or binary data.

*Relationships*

One of the key features of a relational database is the ability to create relationships i.e. associations between tables. This is achieved using keys, which are columns that uniquely identify each record in a table. The most common types of relationships are one-to-one, one-to-many, and many-to-many. These relationships help maintain data consistency and enable efficient retrieval of related information.

*Keys*

Keys are used to establish relationships between tables and ensure data integrity. The primary key uniquely identifies each row in a table, and it must have a unique value for each record. Foreign keys are used to establish relationships between tables by referencing the primary key of another table. By establishing these relationships between tables using keys, data can be easily retrieved and updated across multiple tables. We will cover them in more detail in the next section.

*Indexes*

Indexes are data structures that enhance the performance of queries by providing quick access to specific data within a table. They are created on one or more columns of a table, allowing faster search and retrieval operations based on the indexed values.

*Constraints*

Constraints enforce rules and conditions on the data stored in a database. Common constraints include primary key constraints, foreign key constraints, unique constraints, and check constraints. They ensure data integrity, enforce referential integrity, and prevent the insertion of invalid or inconsistent data.

*Views*

Views are virtual tables derived from the data stored in one or more tables. They are created based on predefined queries and provide a way to present data in a customized or simplified manner without altering the underlying tables. Views can be used for security purposes, data abstraction, or to simplify complex queries.

*Transactions*

Relational databases use *transactions* to keep their state consistent. In the context of database management systems, a transaction is a logical unit of work that represents a series of operations performed on a database as a single indivisible unit. These operations may include inserting, updating, or deleting data from one or more tables in the database. The transactions ensure data consistency and integrity by allowing multiple operations to be executed together. The ACID (Atomicity, Consistency, Isolation, Durability) properties govern the behavior of transactions, which we will cover in more detail in the upcoming section.

These logical core components work together to form the foundation of a relational database schema, providing a flexible and efficient way to store, organize, and retrieve structured data.

**Relational Database Concepts**

Let’s cover basic concepts around relational databases including SQL, ACID, ER Model etc, which form the foundation of modern relational databases.

**SQL**

Relational databases use a structured query language (SQL) to manipulate and retrieve data. SQL allows users to create, modify, and query databases using a set of commands. Some common SQL commands include SELECT, INSERT, UPDATE, and DELETE.

The query language serves as a programming language that establishes the syntax, structure, and semantics governing interactions with the database. It provides a standardized format for storing, accessing, and manipulating data within the database.

The SQL can be categorized in these four types, each serving a distinct purpose:

*Data Definition Language (DDL)*

The Data Definition Language is query language utilized to create and modify the structure and framework of database objects. It encompasses operations such as creating tables, defining indexes, dropping tables, and removing indexes.

*Data Manipulation Language (DML)*

The Data Manipulation Language is query language employed to create, update, delete, and retrieve information from the database. It involves operations such as inserting rows into tables, updating existing rows, deleting rows, and querying data.

*Data Control Language (DCL)*

The Data Control Language is a query language designed to grant or revoke access to entities and operations within the database for clients. It includes commands such as granting privileges to users, revoking privileges, and managing security permissions.

*Transaction Control Language (TCL)*

The Transaction Control Language is query language dedicated to managing and ensuring the consistent execution of a group of database operations as a single unit. It encompasses commands such as committing transactions to make changes permanent or rolling back transactions to discard changes.

Collectively, these query languages form the set of SQL query type, providing the necessary tools to define, manipulate, control access, and manage transactions within a database system.

**ACID**

The ACID model is a set of properties that ensure that database transactions are processed reliably and accurately. ACID stands for:

*Atomicity*

A transaction must be all or nothing. Atomicity ensures that a sequence of operations is treated as a single logical unit of work. Either all the operations within the unit of work are successfully completed, or none of them are applied at all. This guarantee prevents partial updates to the database. When all operations are successfully executed, it is known as a commit. In case of failure, all the operations are rolled back, reverting the database to its original state. Atomicity provides application developers with the confidence that the database will always be in a consistent state, even in the event of failures. It also enables safe retrying of operations without concerns about creating duplicate data.

*Consistency*

A transaction must maintain the consistency of the database. Consistency guarantees ensure that the database transitions from one valid state to another. During the transition, the database enforces rules and constraints defined by the application to maintain data integrity. Consistency is a user-controlled property, meaning that the application must define the valid rules and constraints that lead to a consistent final state. For example, an application may define a rule that the balance of a bank account should always remain positive. The database itself does not enforce these rules but ensures that any changes made to the data adhere to the defined consistency constraints.

*Isolation*

Transactions must be isolated from each other to ensure that they do not interfere with each other. Isolation ensures that concurrent transactions do not interfere with each other, maintaining data integrity and preventing conflicts. Multiple transactions can run simultaneously, and the isolation guarantee dictates how they should interact. If the result of concurrently executing transactions is the same as if they were executed sequentially, then the database supports isolation. Isolation levels, such as Read Uncommitted, Read Committed, Repeatable Read, and Serializable, define the degree of isolation provided by the database. Each level offers different trade-offs between concurrency and data integrity, allowing applications to choose the appropriate level based on their requirements.

*Durability*

Once a transaction is committed, its changes must be permanent and survive any subsequent failures. Durability guarantees ensure that once a transaction is committed, its changes are permanently stored and will survive system failures such as crashes or power outages. Typically, this involves persisting the data to nonvolatile storage like a disk. Durability ensures that critical data remains safe and accessible in the long term. Even in the face of catastrophic events, the database will retain the committed changes and recover them when the system is restored.

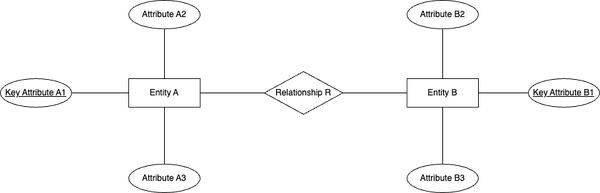
The ACID properties ensure that database transactions are processed in a reliable and consistent manner, even in the presence of failures or concurrent operations. By enforcing these properties, database management systems can maintain the integrity and reliability of the data stored in the database.

In summary, ACID properties provide a set of guidelines for ensuring that database transactions are processed in a reliable and consistent manner, even in the presence of failures or concurrent operations. By adhering to these properties, database management systems can ensure that the data stored in the database remains accurate, reliable, and consistent.

**ER Model**

The ER (Entity-Relationship) model is a conceptual model used in relational database design to describe the relationships between entities (objects or concepts) and their attributes. It provides a graphical representation of the database schema, which can be used to design, communicate, and understand the structure of the database.

The ER model, as shown in [Figure 2-4](https://learning.oreilly.com/library/view/system-design-on/9781098146887/ch02.html#fig_4_er_model_representation) represents entities as rectangles, attributes as ovals, and relationships between entities as diamonds. Relationships can be one-to-one, one-to-many, or many-to-many. The ER model is a useful tool for designing and communicating the structure of a database, as it allows designers to visualize the relationships between entities and the attributes that describe them.



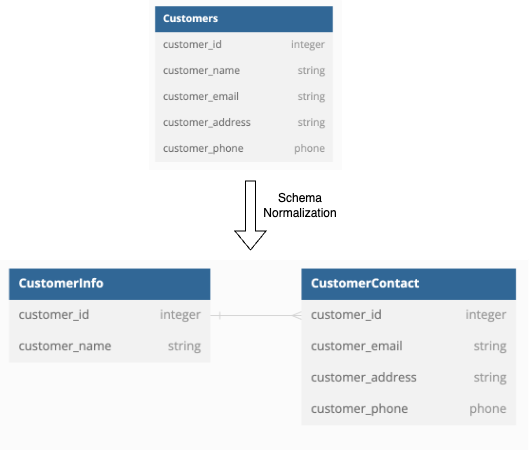
**Figure 2-4. ER Model Representation**

**Schema Normalization**

Schema normalization is the process of organizing data in a database to reduce redundancy and improve data integrity. It involves breaking down a larger table into smaller, more manageable tables, each with its own unique purpose and set of attributes. The goal of schema normalization is to minimize data duplication and ensure that each piece of data is stored in only one place in the database.

Let’s take an example of a table called “Customers” in our Orders database that has the following columns: customer\_id, customer\_name, customer\_email, customer\_address, and customer\_phone. In this table, the customer\_name, customer\_email, customer\_address, and customer\_phone are repeated for each customer.

To normalize this schema as shown in [Figure 2-5](https://learning.oreilly.com/library/view/system-design-on/9781098146887/ch02.html#fig_5_schema_normalization_of_customers_table), we can decompose the “Customers” table into two tables: “CustomerInfo” and “CustomerContact”. The “CustomerInfo” table will contain customer\_id and customer\_name, while the “CustomerContact” table will contain customer\_id, customer\_email, customer\_address, and customer\_phone.



**Figure 2-5. Schema Normalization of Customers Table**

By doing this, we have eliminated the redundancy of customer information in the “Customers” table and stored it only once in the “CustomerInfo” table. This normalization reduces the storage space required for the database, eliminates data inconsistencies that may arise from redundant data, and makes it easier to update customer information without having to change it in multiple places.

**Keys**

Keys are used in relational databases to establish relationships between tables. A key is a column or set of columns that uniquely identifies each row in a table. There are several types of keys:

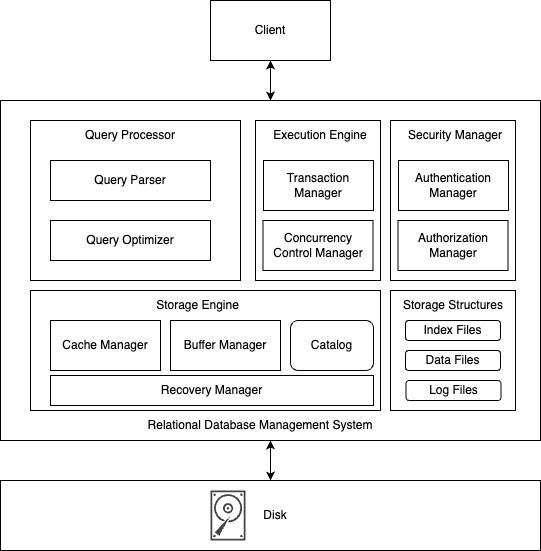
* Candidate key: A candidate key is a column or set of columns in a table that can uniquely identify each record in that table. It is called a “candidate” key because it is a potential primary key “candidate” for the table. For example, in a table of customers, the customer\_email could be a candidate key, as each customer would have a unique email address.
* Primary key: A primary key is a candidate key that has been chosen as the main key for the table. It uniquely identifies each record in the table and is used as a reference by other tables that have relationships with this table. For example, in a table of orders, the Order ID could be the primary key.
* Foreign key: A foreign key is a column or set of columns in one table that refers to the primary key of another table. It is used to establish relationships between tables and ensure data integrity. For example, in a table of orders, there could be a foreign key that references the customer ID in the customers table, to link each order to the customer who ordered it.

In summary, the ER model is a conceptual model used in database design to describe the relationships between entities and their attributes. Schema normalization is the process of organizing data in a database to reduce redundancy and improve data integrity. Keys are used in relational databases to establish relationships between tables, and include candidate keys, primary keys, and foreign keys.

Lets discuss the core components required for a Database Management System architecture. The exact architecture may vary from one implementation to another but generally, the core components remain the same.

**Relational Database Management System Architecture**

This section discusses Relational Database Management System (RDBMS) architecture, which is shown in [Figure 2-6](https://learning.oreilly.com/library/view/system-design-on/9781098146887/ch02.html#fig_6_relational_database_management_system_architecture).



**Figure 2-6. Relational Database Management System Architecture Block Diagram**

Let’s go over the components in detail one by one.

*Query Processor*

The query processor takes the user query and translates it into an execution format suitable for underlying execution engine. It has two main submodules - Query Parser and Query Optimizer.

*Query Parser*

The query processor takes the query given by the user and parses it into Abstract Syntax Tree (AST), which serves as an intermediate representation of the query for execution. It performs parsing, tokenization, syntax validation, semantic analysis, and tree construction in the process of generating AST.

*Query Optimizer*

The query optimizer utilizes the Abstract Syntax Tree (AST) generated by the query parser to generate an optimized plan for executing the user’s query. By considering internal statistics like data cardinality, placement, and costs associated with local and remote execution, the optimizer evaluates multiple execution plans.

*Execution Plan*

An execution plan represents a series of steps organized in a directed dependency graph, which must be executed in a specific order to fulfill the user’s query. Leveraging the internal statistics, the query optimizer selects the most cost-effective execution plan among the alternatives. Subsequently, it forwards this chosen plan to the execution engine for processing.

*Execution Engine*

The execution engine is responsible for executing the query plan generated by the query processor. It interacts with the storage engine to retrieve data, performs any necessary joins, filtering, and sorting operations, and returns the result set to the user. The execution engine is also responsible for orchestrating the execution plan across distributed nodes in a distributed relational database.

*Storage Engine*

The storage engine manages the physical storage and retrieval of data including access and manipulation of data in the database.  It handles tasks such as data page management, file allocation, data compression, and indexing. The storage engine interacts with the execution engine to fetch and store data efficiently.

*Buffer Manager*

The buffer manager handles the management of data buffers in memory. It controls the movement of data between disk and memory, optimizes disk I/O operations, and ensures that frequently accessed data is kept in memory for faster access. The buffer manager minimizes disk access by caching data pages in memory.

*Cache Manager*

The cache manager handles the management and optimization of data caching. It stores frequently accessed data in memory to improve query performance by reducing disk I/O operations. The cache manager ensures that the most frequently used data is readily available in memory for faster retrieval.

*Transaction Manager*

The transaction manager plays a vital role in coordinating and overseeing operations on the data structures within the storage structures module. Its primary responsibility is to ensure that a sequence of operations either executes successfully as a whole or gets rolled back entirely, leaving no partial updates behind.

This crucial guarantee provides end users with the confidence that the database will consistently maintain its integrity before and after executing any database operations. To achieve this, the transaction manager collaborates closely with the concurrency control manager and the recovery manager. By leveraging these components, the transaction manager ensures that the visible data in the database remains consistent and aligns with the expectations of end users.

*Concurrency Control Manager*

The concurrency control manager handles concurrent access to the database by multiple users or transactions. It ensures that transactions are executed in an isolated and consistent manner, preventing conflicts and maintaining data integrity. It manages locking, transaction isolation levels, and conflict resolution mechanisms.

*Recovery Manager*

The recovery manager ensures the durability and reliability of the database in the event of failures or crashes. It manages transaction logging, checkpointing, and crash recovery mechanisms to maintain data integrity and consistency. The recovery manager ensures that the database can be restored to a consistent state after a failure.

The recovery manager ensures durability by maintaining an immutable data structure known as a log file within the DBMS. This log file diligently records every write operation applied to the database, ensuring its ability to facilitate recovery processes. Essentially, the recovery manager acts as a reliable and persistent intermediate storage for all write requests.

Each written page is meticulously preserved in the primary memory, designated as a “dirty page.” To ensure durability, the recovery manager batches all the dirty pages and asynchronously synchronizes them with the disk. Once the dirty pages have been successfully flushed to disk, they are considered “clean pages.” In addition, before a write request is acknowledged as completed by the client, it is also appended to a disk-resident append-only data structure. This additional step provides a safeguard against data loss of dirty pages in the event of a crash or system restart.

The log file serves as a valuable checkpoint, allowing the recovery process to restore data for the dirty pages. On system restart, the operating system takes responsibility for flushing all the remaining dirty pages to disk and discarding any uncommitted transactions. This ensures that the system begins in a clean state, ready to process subsequent writes and reads, without the risk of incomplete or inconsistent data.

*Security Manager*

The security manager is responsible for enforcing data security and access controls. It authenticates users, manages user permissions and privileges, and ensures data confidentiality, integrity, and availability. The security manager protects the database from unauthorized access and maintains data privacy.

*Catalog*

The catalog, also known as the data dictionary or metadata repository, stores the metadata about the database schema, tables, columns, indexes, constraints, and other database objects. It provides information about the structure and organization of the database, allowing the RDBMS to interpret and manipulate the data accurately.

These core components work together to provide the necessary functionality for managing and manipulating data in a RDBMS. They ensure efficient query processing, data storage, concurrency control, data recovery, and data security.

In the next section, we will explore the different ways of optimizing query performance in RDBMS using index types, including primary and secondary indexes, SQL tuning, Denormalization and Query Federation.

**Optimizing Relational Databases**

Optimizing relational databases is an essential task for any database administrator or developer to improve the performance of the queries that run on top of it. Let’s cover various ways to optimize databases in detail.

**Indexes**

One way to optimize SQL queries is by using indexes. An index is a data structure that improves the speed of data retrieval operations on a database table. By creating indexes on frequently used columns, the optimizer can quickly find the data needed to execute the query, resulting in faster query execution times instead of performing full table scan.  Columns that you are querying (SELECT, GROUP BY, ORDER BY, JOIN) could be faster with indices. Indices are usually represented as self-balancing B-tree that keeps data sorted and allows searches, sequential access, insertions, and deletions in logarithmic time. Following are the two types of indexes, which can be created in a RDBMS:

*Primary Indexes*

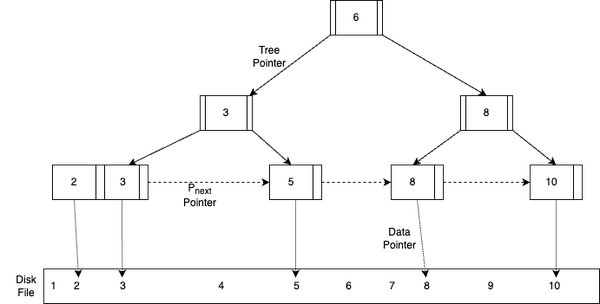
A primary index is an index that is created on a table’s primary key. The primary key is a column or set of columns that uniquely identify each row in the table. By creating a primary index on the primary key, the database can quickly find the location of a specific row in the table, resulting in faster data retrieval times.

*Secondary Indexes*

A secondary index is an index that is created on a non-primary key column or set of columns. A secondary index can be used to improve the performance of queries that involve filtering or sorting data based on a specific column or set of columns. For example, if a query frequently filters data based on a customer’s phone number, creating a secondary index on the phone number column can significantly improve the query’s performance.

B+ trees are widely used in RDBMS as an efficient indexing structure for facilitating fast data retrieval and efficient query processing. The B+ tree data structure provides efficient key-based searching and range queries on large amounts of data.

A B+ tree, illustrated in [Figure 2-7](https://learning.oreilly.com/library/view/system-design-on/9781098146887/ch02.html#fig_7_b_tree_data_structure_representation) is a self-balancing tree data structure that stores data in a sorted manner. It consists of internal nodes and leaf nodes, where leaf nodes contain the actual data records or pointers to the data records. Each node in the B+ tree has a fixed number of keys and pointers. The keys in the internal nodes act as separators, guiding the search process, while the leaf nodes store the data records or pointers to the data records in sorted order.



**Figure 2-7. B+ tree data structure representation**

When an index is created on a column, a B+ tree index is constructed, where each node in the tree corresponds to a range of values from the indexed column. The leaf nodes of the B+ tree contain the values of the indexed column along with the corresponding pointers to the actual data records. B+ trees can also be used to create multi-column indexes in RDBMS. In this case, each node in the B+ tree contains multiple keys, allowing for efficient searching and querying based on multiple columns. Multi-column indexes are beneficial for queries that involve complex conditions or join operations on multiple columns.

Let’s understand the effectiveness of B+ tree in different types of queries, scans and operations:

*Efficient Searching and Range Queries*

B+ trees provide efficient key-based searching and range queries, which are crucial for improving query performance in RDBMS. When a query involves searching for a specific value or a range of values, the B+ tree index allows the database system to quickly navigate the tree to find the desired data records. The balanced nature of the B+ tree ensures that the height of the tree remains relatively small, resulting in faster search operations with logarithmic time complexity.

*Sequential and Range Scans*

B+ trees also enable efficient sequential and range scans in RDBMS. The leaf nodes of the B+ tree are linked together, allowing for efficient scanning of the entire index or a specific range of values. Sequential or range scans are often performed for queries involving sorting, aggregations, or when a subset of data needs to be retrieved based on specific conditions.

*Update and Insert Operations*

B+ trees efficiently handle update and insert operations in RDBMS. When a new record is inserted, the B+ tree index is updated in a way that ensures the tree’s balance and sorted order are maintained. Similarly, when a record is updated, the corresponding entry in the B+ tree index is adjusted accordingly. This ability to handle dynamic updates and inserts without significant performance degradation makes B+ trees suitable for transactional systems.

However, using a lot of indices comes with its own caveats as placing an index can keep the data in memory, requiring more space. Writes could become slower since the index also needs to be updated. When loading large amounts of data, it might be faster to disable indices, load the data, then rebuild the indices.

**SQL Tuning**

As prescribed under “Prescription of Performance : Metrics Don’t Lie” in Chapter 1, to improve the performance of SQL queries, it is first important to benchmark and profile your queries to uncover bottlenecks. Benchmarking will involve simulating high-load and high-throughput situations, while profiling will require measuring the p9s of the query and analyzing slow query log to identify performance issues. Based on these metrics, SQL tuning involves improving SQL queries to improve their performance by removing the bottlenecks and optimizing query plans.

One way to optimize SQL query is by minimizing large write operations as performing operations such as writing, modifying, deleting, or importing extensive amounts of data can have a significant impact on the performance of queries. These operations may even result in table blocking when tasks involve updating and manipulating data, adding indexes or check constraints to queries, processing triggers, and similar actions. Moreover, the act of writing a substantial volume of data will inevitably lead to an increase in the size of log files.

To optimize SQL performance, another technique is to schedule query execution during off-peak hours. This approach is particularly beneficial when dealing with multiple SELECT queries involving large tables or executing complex queries with nested subqueries, looping queries, and the like. When executing resource-intensive queries in a database, RDBMS applies locks to the tables involved, preventing simultaneous access by different transactions. Consequently, other users are unable to work with those tables, leading to limited access to specific data. Running heavy queries during peak times not only strains the server but also restricts other users’ data access.

Adding multiple tables to a query and performing joins can potentially burden the query and lead to performance issues. Moreover, when dealing with a large number of tables to retrieve data from, it may result in an inefficient execution plan. To achieve efficient query plans, JOIN elimination is one of the many techniques employed. By dividing a single query into several separate queries that can be joined later, unnecessary joins, subqueries, and tables can be eliminated. This approach helps streamline the query and enhance its performance by removing redundant and extraneous elements.

Enhancing RDBMS performance and optimizing SQL queries are critical aspects for both database developers and administrators. They must meticulously evaluate various factors such as the selection of specific operators, the number of tables involved in a query, the query’s size, its execution plan, statistics, resource allocation, and other performance metrics. These considerations play a pivotal role in determining whether query performance improves or deteriorates. By carefully analyzing and addressing these factors, developers and administrators can effectively tune and improve query performance.

**Denormalization**

In most systems, read operations significantly outnumber write operations by a ratio of 100:1 or even 1000:1. Performing a complex database join during a read can be resource-intensive, particularly due to disk operations.

Denormalization is the technique aimed at improving read performance, although it may come at the cost of reduced write performance. It involves duplicating data across multiple tables to avoid costly joins. Some RDBMS offer materialized views, which handle the task of storing redundant information and maintaining consistency among redundant copies.

When data is distributed through methods such as federation and sharding, managing joins across different data centers adds further complexity. Denormalization can help alleviate the need for complex joins in such scenarios.

However, denormalization also has its disadvantages due to data duplication, leading to redundancy. Maintaining consistency among redundant copies requires the use of constraints, which further, adds complexity to the database design. A denormalized database under heavy write load may perform worse compared to its normalized counterpart.

Overall, denormalization can provide performance benefits for read-heavy workloads, but it introduces trade-offs in terms of data redundancy and increased complexity in database management.

**Query Federation**

Query federation is a technique that involves splitting a large query into smaller queries that can be executed independently on different database servers. This technique requires schema federation, i.e. functional partitioning of the database across multiple database servers. By executing smaller queries on different servers, the overall query execution time can be reduced, resulting in faster query results. Query federation is useful for optimizing queries that involve large amounts of data or complex joins.

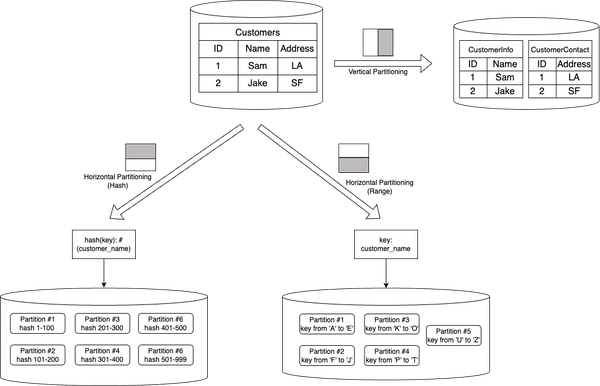
In conclusion, optimizing relational databases using indexes is a critical task for any database administrator or developer. By using these above techniques, database administrators can improve the performance of their databases and provide faster and more efficient access to data. Next, we cover the techniques used to scale relational databases in practice including partitioning, sharding and replication.

**Scaling Relational Databases**

As businesses grow and their data needs increase, it becomes necessary to scale their databases to handle the increased workload. Scaling refers to the process of increasing the capacity of a database to accommodate more data, users, and transactions. In this section, we will explore scaling relational databases using partitioning, sharding, and replication.

**Partitioning**

Partitioning is the process of dividing a large database table into smaller, more manageable parts called partitions. Each record within the database is assigned to a specific partition, ensuring that every record belongs to one and only one partition. Each partition operates as an independent database, capable of executing read and write operations autonomously. As a result, database queries can be directed towards a single partition to focus on specific data or distributed across multiple partitions for broader processing. The client can either direct the query to a specific partition or to a coordinator node, which then forwards the query to the right set of partitions, thus orchestrating the partitions. There are two types of partitioning as shown in [Figure 2-8](https://learning.oreilly.com/library/view/system-design-on/9781098146887/ch02.html#fig_8_partitioning_approaches_in_database): vertical and horizontal, with two possible approaches to horizontal partitioning:



**Figure 2-8. Partitioning approaches in database**

*Vertical partitioning*

Vertical partitioning involves splitting a table by columns. For example, a customer table might be split into two tables, one containing customer information and another containing customer contact.

*Horizontal partitioning*

Horizontal partitioning involves splitting a table by rows. For example, a large customer table might be split into smaller tables, each containing a specific range of customer records based on their last name or zip code. Horizontal partitioning can again have two approaches: Hash Partitioning and Range Partitioning.

*Hash Partitioning*

The partition by key hash strategy involves generating a hash of the key and evenly distributing it among the partitions. This approach helps avoid data skew and eliminates hot spots within the system.

In this strategy, a hash function (such as MD5 or SHA-256) is applied to the input key of the table. The hash ranges are then divided into buckets, with each bucket assigned to a specific partition. It’s important to note that a single host instance can accommodate multiple partitions.

For this strategy to work effectively, the hash function must be deterministic, meaning that the same key should always produce the same hash value and be directed to the same partition. Additionally, each key should resolve to a unique hash value, ensuring a balanced distribution of data across the partitions.

The advantage of hash partitioning is that it eliminates the hot spots and skew partition problem by unique deterministic hashing strategy. However, since the partitioning is on the key’s hash, this strategy can’t do range queries. We need to query all the partitions, if we want to fetch a set of keys, making the DBMS do a *scatter and gather pattern*.

*Range Partitioning*

In range partitioning, we partition a continuous range of keys into separate buckets, which are then assigned to specific partitions. It is important to note that a single host instance has the capability to accommodate multiple partitions. The key range assigned to each bucket may or may not be continuous, allowing flexibility in the distribution of data.

Within each partition, the keys are stored in sorted order. This organization of data enables efficient range scan queries, as it simplifies the process of retrieving data within specific key ranges. By maintaining sorted order within each partition, the system can swiftly access and retrieve data that falls within a given range.

One of the key benefits of range partitioning is that individual partitions store keys in a sorted order, facilitating efficient range scans. For instance, in the figure, it becomes easy to retrieve all customers having email starting with ‘P’ in lexicographic order. Additionally, the range partitioning strategy is relatively straightforward to implement. However, if the access patterns on partitions are uneven or unfair, certain partitions may end up with more data or queries than others, leading to skewed partitions and placing a heavier load on specific partitions, potentially causing congestion. Such imbalanced partitions, experiencing a disproportionately high workload, are referred to as hot spots. To illustrate, in [Figure 2-8](https://learning.oreilly.com/library/view/system-design-on/9781098146887/ch02.html#fig_8_partitioning_approaches_in_database), if a significant majority of customers have email begining with ‘A', it can result in the choking of Partition 1 due to an overwhelming amount of data and queries concentrated in that partition.

Partitioning a database offers advantages for both large datasets as well as enables high throughput capability. Partitioning enables the distribution of data across multiple machines via sharding, allowing for the handling of datasets that exceed the capacity of a single machine. By spreading the data across multiple partitions, the database can accommodate and manage large-scale data requirements.

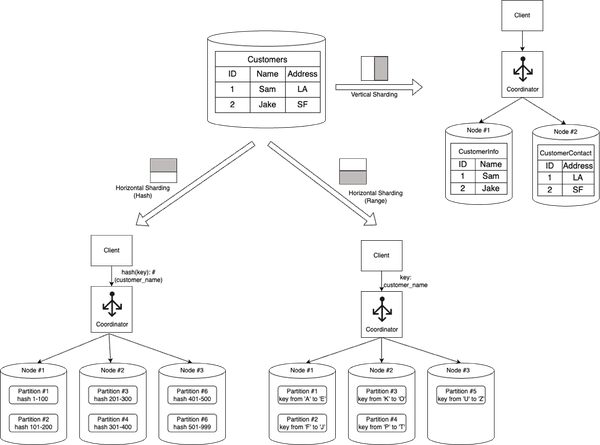
With data distributed across multiple partitions, read and write queries can be processed independently by each partition. This parallel processing capability allows the database to handle a higher overall throughput compared to what a single machine can handle. By leveraging the collective power of multiple machines, partitioning enhances the system’s ability to handle concurrent queries and transactions efficiently.

Thus, partitioning can improve query performance by reducing the amount of data that needs to be scanned to execute a query. This results in faster query response times and better scalability.

**Sharding**

Sharding is the process of distributing a large database across multiple servers. Each server contains a subset of the data, and queries are distributed across all servers. Sharding is useful for scaling databases that have become too large to be managed on a single server. For example, in a customer database, as the number of customers grows, more shards are added to the cluster to accommodate the increasing load.

There are two types of sharding as shown in [Figure 2-9](https://learning.oreilly.com/library/view/system-design-on/9781098146887/ch02.html#fig_9_sharding_approaches_in_database): vertical and horizontal, with different possible approaches to horizontal partitioning, including hash-based sharding, range-based sharding, and round-robin sharding.



**Figure 2-9. Sharding approaches in database**

Sharding offers similar advantages to federation, including reduced read and write traffic, decreased replication, and improved cache utilization. It also helps reduce the size of indexes, leading to faster query performance. In the event of a shard failure, the other shards remain operational, although implementing replication is crucial to prevent data loss. Sharding eliminates the need for a central master for write serialization, enabling parallel writes and increasing overall throughput.

Common approaches to shard a customer table include using the customer’s last name, initial, or geographic location. However, sharding comes with its disadvantages. Application logic needs to be updated to handle shard-specific operations, potentially resulting in complex SQL queries. Data distribution can become unbalanced, particularly if a shard has a subset of power customers, leading to increased load on that particular shard. Rebalancing shards adds complexity, although using a consistent hashing-based sharding function can minimize data transfer. Joining data from multiple shards becomes more challenging, and overall, sharding introduces additional hardware requirements and increased system complexity.

**Replication**

Replication is the process of copying data from one database server to another. Every node that stores a copy of the data is called a replica. Replication is a key feature in distributed databases that offers several advantages, enhancing availability, load distribution, and reducing latency. By maintaining multiple copies of data across different host machines, replication provides the following benefits:

*High Availability*

Replication ensures high availability by storing data copies on multiple host machines. In the event of a failure or downtime of one host machine, the database can seamlessly redirect read and write operations to other live machines that hold replicated data. This fault tolerance mechanism prevents service disruptions and ensures continuous access to data, enhancing the overall availability of the system.

*Load Distribution*

With replication, the database can distribute read and write queries across multiple host machines. This load distribution strategy prevents overburdening of individual machines, thereby improving the overall system performance and scalability. By spreading the workload, replication enables efficient utilization of computing resources and better handling of concurrent user requests, resulting in enhanced throughput and responsiveness.

*Reduced Latency*

Replicating data across geographically distributed host machines allows for placing the replicated copies closer to end users. This proximity reduces the network latency experienced by users when accessing the database. By minimizing the distance between the data and the user, replication improves response times and enhances the user experience, especially in scenarios where low latency is critical, such as real-time applications or distributed systems with users located in different regions.

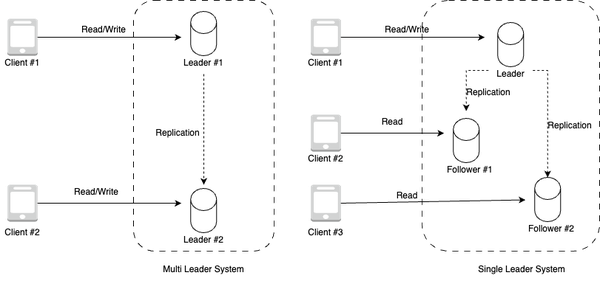
*Disaster Recovery and Data Resilience*

Replication serves as a foundation for disaster recovery strategies. By maintaining multiple copies of data, distributed databases can withstand catastrophic events, hardware failures, or natural disasters. In case of data loss or unavailability on one host machine, the replicated copies can be used for data recovery and system restoration. Replication ensures data resilience and minimizes the risk of data loss, contributing to the overall reliability and robustness of the database.

*Scalability and Performance*

Replication plays a crucial role in scaling distributed databases. As the data size or user load increases, additional host machines can be added to the system, each hosting a replicated copy of the data. This horizontal scaling approach allows the database to handle larger workloads and accommodate growing user demands. By distributing data and queries, replication contributes to improved system performance, enabling efficient parallel processing and reducing response times.

There are two types of replication for RDBMS: Single-leader and Multi-leader, both of which are illustrated in [Figure 2-10](https://learning.oreilly.com/library/view/system-design-on/9781098146887/ch02.html#fig_10_replication_approaches_in_database).



**Figure 2-10. Replication approaches in database**

*Single-leader replication*

In single leader replication, one database server serves as the leader, and the other server(s) serve as followers. The leader server handles all write operations, while the follower servers handle read operations. Changes made to the reader server are replicated to the follower servers, ensuring that all servers contain the same data either synchronously or asynchronously. This type of replication is useful for scaling read-heavy databases.

*Multi-leader replication*

In multi-leader replication, each database server can both read and write data. Changes made to one server are replicated to the other server either synchronously or asynchronously, ensuring that all servers contain the same data. This type of replication is useful for scaling databases that require high availability.

For the above replication types, replication can be done on follower nodes either as full replication, snapshot based replication, transactional replication (i.e. replicating the transactional updates) or key based incremental replication (i.e. scaning the database for modified keys and only replicating data for those keys). The replication can be done either synchronously or asynchronously, each having its own caveats

In distributed databases, synchronous replication provides a mechanism for replicating data from a leader replica to follower replicas using synchronous communication, keeping data always up-to-date across all replicas. By maintaining synchronous replicas that are always in sync with the leader replica, this replication mechanism offers several advantages:

*Consistent Writes*

With synchronous replication, the client considers a write operation successful only when it is acknowledged by both the leader replica and the synchronous follower replica. This strict synchronization ensures that the data remains consistent across replicas. By waiting for confirmation from both replicas, the system guarantees that the write has been durably committed and is available for subsequent read operations. This consistency is vital for applications that require strict data integrity and accuracy.

*Immediate Failover*

One significant advantage of synchronous replication is its ability to facilitate immediate failover in the event of a leader replica crash. Since the synchronous replicas are always up-to-date with the leader, any of these replicas can be immediately promoted as the new leader without any data loss. This seamless transition ensures continuous availability of the system even in the face of failures. By eliminating downtime and minimizing data loss, synchronous replication enhances the overall resilience and reliability of the distributed system.

*Data Durability*

Synchronous replicas guarantee that data is durably stored across multiple replicas. When a write operation is confirmed by both the leader and synchronous follower replicas, it ensures that the data is safely persisted on multiple machines. This redundancy provides data durability, protecting against data loss in the event of a replica failure or system crash. By maintaining multiple synchronized copies, synchronous replication ensures that critical data remains intact and recoverable, enhancing the overall data resilience of the system.

*Consistency in Read Operations*

Synchronous replication not only ensures consistency in write operations but also in read operations. As the synchronous replicas are always up-to-date with the leader, read operations can be performed on any of the replicas with the guarantee of accessing the most recent and consistent data. This feature enables load balancing and improves the system’s ability to handle read-intensive workloads.

In summary, synchronous replication in distributed databases provides the benefits of consistent writes, immediate failover, data durability, and consistency in read operations. By maintaining synchronous replicas that are always in sync with the leader, this replication mechanism ensures data integrity, continuous availability, and reliable access to up-to-date information. Synchronous replication is particularly valuable in scenarios where strict consistency and high availability are paramount, such as in financial systems, real-time applications, and mission-critical environments.

In distributed databases, asynchronous replication provides a mechanism for replicating data from a leader replica to asynchronous replicas. This replication approach offers specific characteristics and considerations:

*Near Real-Time Updates*

Asynchronous replicas apply changes from the leader replica in near real-time, albeit with a potential delay. This means that the asynchronous replicas may not be immediately up-to-date with the latest changes made on the leader. The time lag between the leader and asynchronous replicas results in a temporary inconsistency in data across the replicas. However, over time, the asynchronous replicas converge as they catch up with the leader’s updates.

*Data Lag and Potential Staleness*

Due to the asynchronous nature of replication, the asynchronous replicas can lag behind the leader replica. The extent of this lag depends on various factors such as network conditions, system load, and the volume of changes being replicated. As a result, the asynchronous replicas may not reflect the most recent state of the data. This potential staleness can impact applications or use cases that require access to the most up-to-date information. It’s essential to consider this trade-off between near real-time updates and potential data lag when employing asynchronous replication.

*Risk of Data Loss*

Because asynchronous replicas can be lagging or stale, promoting an asynchronous replica as the new leader in the event of a leader replica crash can introduce the risk of data loss. Since the asynchronous replica may not have received or applied all the changes made on the previous leader, promoting it prematurely can result in missing or inconsistent data. To mitigate this risk, careful considerations and measures, such as monitoring the replication lag and ensuring that the asynchronous replica has caught up sufficiently, should be taken before promoting it as the new leader.

*Scalability and Performance*

Asynchronous replication is often favored in distributed systems that prioritize scalability and performance over strict consistency. By allowing some flexibility in the replication process and tolerating temporary data inconsistencies, asynchronous replication can handle high write throughput and accommodate systems with large-scale deployments. This approach enables the system to distribute the workload and scale horizontally while maintaining acceptable response times.

In summary, asynchronous replication in distributed databases provides near real-time updates but introduces the possibility of data lag and potential staleness. While it offers scalability and performance advantages, the risk of data loss exists when promoting an asynchronous replica as a new leader. Careful planning, monitoring, and appropriate fallback mechanisms are necessary to ensure data integrity and minimize the impact of data inconsistencies when employing asynchronous replication.

Replication in distributed databases offers numerous benefits, including high availability, load distribution, reduced latency, disaster recovery capabilities, scalability, and enhanced performance. By leveraging replication strategies, organizations can ensure robustness, fault tolerance, and optimized access to data in distributed environments.

Scaling distributed relational databases is an important task for businesses that need to handle increasing amounts of data and traffic. Partitioning, sharding, and replication are all useful techniques for scaling databases. Partitioning can improve query performance by reducing the amount of data that needs to be scanned to execute a query. Sharding is useful for scaling databases that have become too large to be managed on a single server. Replication is useful for scaling databases that require high availability or have heavy read traffic.

To conclude the chapter, let’s go over the most popular open-source RDBMS choices in the next section.

**Open-source Relational Database Systems**

Open-source RDBMS are popular choices for businesses and developers due to their affordability, flexibility, and community support. Two widely used open-source RDBMS are MySQL and PostgreSQL. In this section, we will compare and contrast MySQL and PostgreSQL and explore their similarities and differences.

*MySQL*

[MySQL](https://dev.mysql.com/doc/) is an open-source RDBMS that was first released in 1995. It is one of the most popular RDBMS in the world, with a large user community and a broad range of features. MySQL is known for its speed and scalability, making it an ideal choice for businesses that need to handle high volumes of data and traffic.

Features of MySQL include:

* Supports SQL and non-SQL queries
* Scalable architecture
* High performance
* Simple and easy to use
* Good for read-heavy workloads
* Strong support for clustering and replication

*PostgreSQL*

[PostgreSQL](https://www.postgresql.org/docs/) is an open-source RDBMS that was first released in 1996. It is known for its robustness, reliability, and feature-richness. PostgreSQL is a popular choice for businesses that require advanced data management capabilities and powerful query processing.

Features of PostgreSQL include:

* Supports SQL and non-SQL queries
* ACID-compliant transactions
* Strong support for stored procedures and triggers
* Supports JSON and XML data types
* Good for write-heavy workloads
* Offers advanced data management features

Both MySQL and PostgreSQL, as compared in Table 2-1, are open-source RDBMS, are ACID-complaint and support SQL and non-SQL queries. Both MySQL and PostgreSQL offer strong support for replication and clustering. However, MySQL has a simpler and more straightforward syntax, while PostgreSQL offers more advanced features and capabilities. MySQL is better suited for read-heavy workloads, while PostgreSQL is better suited for write-heavy workloads.

|  |  |  |
| --- | --- | --- |
| Property | MySQL | PostgreSQL |
| Introduction | MySQL is open-source RDBMS. | PostgreSQL is an object-relational database management system (ORDBMS). It provides all the facilities of RDBMS with additional support of object oriented concepts like classes, objects and inheritance. |
| Datatype Support | Supports standard SQL types. | Supports the standard SQL types along with many advanced types such as array, jsonb, and user- defined type. |
| JSON Support | Supports JSON documents to be stored in a column by converting to an internal format that permits quick read access to document elements.  JSON columns cannot be indexed directly. To create an index that references such a column indirectly, you can define a generated column that extracts the information that should be indexed, then create an index on the generated column. | Support two data types related to json.  Json: This stores an exact copy of the input text which processing functions must reparse on each execution.  Jsonb: This is stored in a decomposed binary format that makes it slightly slower to input due to added conversion overhead, but significantly faster to process, since no reparsing is needed. It also supports indexing which can be a big advantage. |
| Indexes | Types of indexes include primary key, foreign key, unique Index, single column, multi column index(upto 16 columns). spatial indexes. | Types of indexes include primary key, foreign key unique index, single column, multi column index(upto 32 columns)  Besides that, it also support following two types.  Expression indexes: can be created with an index of the result of an expression or function, instead of simply the value of a column.  Partial indexes: index only a part of a table. |
| Replication | MySQL replication is one-way asynchronous replication where one server acts as a master and others as slaves You can replicate all databases, selected databases or even selected tables within a database. | PostgreSQL has synchronous replication (called 2-safe replication), that utilizes two database instances running simultaneously where the master database is synchronized with a slave database. Unless both databases crash simultaneously, data won’t be lost. |
| Performance | MySQL implements concurrent connections by spawning a thread-per-connection This is relatively low overhead. | Postgres, however, uses a process-per-connection design. This is significantly more expensive than a thread-per-connection design. Postgres also seems to have poor support for handling large connection counts even when there is sufficient memory available. |
| Speed | By not including certain SQL features, MySQL stays light to prioritize speed and reliability. MySQL’s speed is specially apparent, when it comes to highly concurrent read-only operations. | Postgres supports query plans that can leverage multiple CPUs in order to answer queries with greater speed. This, coupled with strong support for multiple concurrent writes, makes it a great choice for comple operations like data warehousing and online transaction processing (OLTP). |
| Table 2-1. MySQL vs PostgreSQL | | |

In conclusion, both MySQL and PostgreSQL are powerful open-source RDBMS that offer a broad range of features and capabilities. MySQL is known for its speed and scalability, while PostgreSQL is known for its robustness and advanced data management capabilities. When choosing between MySQL and PostgreSQL, it’s important to consider the specific needs of your business and the requirements of your application.

**NOTE**

AWS Relational Database Services (RDS) offers multiple options  by providing various database engine versions (commonly referred to as “flavors”), instance classes, and storage types. These flavors are essentially different configurations of managed database engines, each optimized for specific use cases and workloads. AWS RDS supports several popular relational database engines, including MySQL, PostgreSQL, Oracle, SQL Server, and MariaDB. We will cover AWS RDS in more detail in Chapter 10 - AWS Storage Services.

**Conclusion**

This chapter has provided a comprehensive exploration of relational databases and their underlying concepts, architecture, and strategies for scalability. We began by delving into various types of storage mechanisms, including file, block, and object stores, laying the foundation for understanding how data is managed and accessed. From there, we transitioned into a thorough examination of relational databases, elucidating the key principles that govern their design and operation.

The focal point of the chapter revolved around addressing the challenges of database scalability. We uncovered a range of advanced techniques, such as partitioning, indexing, replication, federation, sharding, and denormalization, each offering unique solutions to accommodate growing datasets and increasing user demands. By exploring these strategies, you have gained valuable insights into how to effectively optimize performance and maintain the integrity of relational databases as their usage expands.

To further enrich your understanding, we introduced two prominent open source databases, MySQL and PostgreSQL, allowing you to acquaint yourselves with practical implementations of the concepts covered. These databases serve as powerful tools, showcasing the real-world application of the theoretical knowledge presented in the chapter.

As we conclude this chapter on relational databases, we open the door to the exciting realm of non-relational databases. The next chapter will embark you on an exploration of alternative database paradigms that have gained prominence in recent years. Non-relational databases, with their diverse models and unique features, present a compelling alternative to traditional relational systems. By delving into this subject, you will broaden your horizons and deepen your understanding of the evolving landscape of database technologies.