**Chapter 3. Non-relational 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 3rd 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 the ever-evolving landscape of modern data management, traditional relational databases have long been the stalwarts of structured data storage and retrieval. However, the rise of new technologies, diverse data formats, and the need for high-performance, scalable solutions has given birth to a new class of databases known as non-relational or NoSQL databases. These databases have gained popularity for their ability to handle the challenges posed by today’s data-intensive applications and their capacity to adapt to various data models, while offering unprecedented scalability and performance.

Traditionally, relational databases have been the cornerstone of data management, providing a structured and standardized approach to organizing and retrieving information. However, as the digital universe continues to expand exponentially, traditional relational models face inherent limitations when confronted with the demands of modern web applications, real-time analytics, and large-scale distributed systems.

Non-relational databases break away from the rigid constraints of the relational paradigm and introduce novel data models and storage mechanisms that challenge the status quo. These databases prioritize scalability, fault tolerance, and low-latency access, enabling organizations to effectively handle massive volumes of data and support dynamic, rapidly evolving data requirements.

In this chapter, we dive deep into the realm of non-relational databases. We will unravel the fundamental principles, key concepts, and various types of NoSQL databases, shedding light on their strengths, weaknesses, and ideal use cases. Our exploration will encompass a range of popular NoSQL database models, including document stores, key-value stores, columnar databases, and graph databases, each tailored to address specific data management challenges. We’ll also discuss their architecture and how leaderless architecture supports scaling using quorum, optimistic replication, consistent hashing, and hinted hand-off.

Throughout this chapter, we will delve into the intricacies of data modeling, query languages, and consistency models associated with non-relational databases. We will examine the core architectural principles that underpin their design, such as horizontal scalability, distributed data storage, and decentralized control. Additionally, we will explore the trade-offs that come with embracing non-relational databases, including data consistency, data integrity, and the impact on application development.

By the end of this chapter, you will have gained a comprehensive understanding of non-relational databases, empowering you to make informed decisions when choosing the right data storage solution for your organization’s unique needs. Whether you are a seasoned data professional or a curious enthusiast, prepare to embrace the world of non-relational databases and discover the remarkable possibilities they offer in the era of big data and beyond.

**NOTE**

AWS offers a range of non-relational database services, such as Amazon DynamoDB, Amazon DocumentDB, Amazon Neptune, Amazon ElasticCache, Amazon Opensearch, Amazon Keyspaces and more, to meet customer requirements for different business use-cases, which we will cover in more detail in Chapter 10 - AWS Storage Services.

Let’s cover basic concepts in the next section around non-relational databases including BASE, schema-less design, horizontal scalability, high availability etc, which form the foundation of modern non-relational databases.

**Non-relational Database Concepts**

Some of the fundamental concepts that characterize non-relational databases include:

*Schema Flexibility*

Unlike relational databases that enforce a fixed schema, non-relational databases allow for dynamic and flexible schema design. This means that data can be stored without the need to pre-define a strict structure, making it easier to accommodate varying data formats and evolving application requirements.

*Data Models*

Non-relational databases support various data models, each optimized for specific use cases. The most common data models include:

*Document Stores*

These databases store and manage data in flexible, semi-structured documents, often in formats like JSON or BSON. This approach is ideal for applications dealing with complex or dynamic data.

*Key-Value Stores*

These databases store data in a simple key-value format, making them highly efficient for read and write operations. They are commonly used for caching and high-speed data retrieval.

*Column-Family Stores*

These databases organize data into column families, allowing efficient storage and retrieval of large volumes of data, especially in analytical and data warehousing scenarios.

*Graph Databases*

Graph databases focus on relationships between data entities, making them well-suited for applications that require complex querying and analysis of interconnected data.

*Scalability*

Non-relational databases are designed to scale horizontally, distributing data across multiple nodes or servers. This enables them to handle massive datasets and high levels of concurrent traffic. Scaling is achieved through techniques like sharding and replication, allowing applications to grow seamlessly as demand increases.

*High Availability and Fault Tolerance*

Many non-relational databases prioritize high availability and fault tolerance. They are often designed to handle hardware failures, network partitions, and other disruptions without compromising data integrity or accessibility.

*BASE*

In the realm of non-relational databases, the acronym BASE stands for Basically Available, Soft state, Eventually consistent. BASE is an alternative approach to data consistency and availability as per CAP theorem, compared to the ACID (Atomicity, Consistency, Isolation, Durability) properties traditionally associated with relational databases. BASE is particularly relevant in distributed and large-scale systems where high availability and scalability are key requirements, often at the expense of strict strong consistency.

**NOTE**

Please refer to Chapter 1 - System Design Tradeoffs and Guidelines to read about CAP theorem, a trade-off which naturally arises in distributed systems. Also, refer to Chapter 2 - Storage Types and Relational Stores to read about ACID in more detail, which is fundamental property in relational databases.

*Basically Available*

The “Basically Available” aspect of BASE emphasizes that a non-relational database should always ensure some level of availability, even in the face of network partitions, hardware failures, or other issues. This means that the database system is designed to respond to client requests even if it means sacrificing some level of consistency in the data. Availability is a critical requirement for modern applications that cannot afford downtime or unresponsiveness.

*Soft State*

In a non-relational database system following the BASE principles, the concept of “Soft State” is adopted. Soft state implies that the state of the system can change over time due to various factors, such as node failures, network delays, or concurrent updates. This contrasts with the traditional ACID principle of maintaining a hard or rigid state consistency at all times. In a BASE system, it’s acknowledged that the system’s state might be fluid, and applications need to be designed to handle such variability gracefully.

*Eventually Consistent*

Perhaps the most distinct characteristic of BASE is the principle of “Eventually Consistent.” Unlike ACID, which enforces strict consistency at all times, BASE allows for a temporary lack of consistency between replicas in a distributed system. In other words, it recognizes that updates to the database may take some time to propagate across all nodes, and there might be a brief period during which different nodes may have slightly different views of the data. However, over time, as the system resolves conflicts and synchronizes updates, the data will eventually converge to a consistent state.

BASE principles are particularly suitable for scenarios where rapid scalability and high availability are paramount, such as in modern web applications, real-time analytics, and large-scale data processing. It’s important to note that while ACID properties provide strong guarantees of data integrity and consistency, they can potentially hinder performance and scalability in distributed environments. BASE, on the other hand, offers a trade-off between consistency and availability, allowing applications to maintain responsiveness and continue functioning in the face of network partitions or hardware failures.

It’s worth mentioning that the choice between ACID and BASE depends on the specific requirements of an application. Some applications, like financial systems or traditional relational databases, may prioritize strict consistency provided by ACID, hence the need to use relational databases. Meanwhile, others, like social media platforms or content delivery networks, may benefit more from the availability and scalability offered by BASE principles.

Let’s go over the different types of non-relational databases based on data models discussed above in detail, starting with key value databases.

**Key Value Databases**

Key-value databases are a type of non-relational database that offer a simple yet powerful data model, enabling efficient storage and retrieval of data based on unique keys using distributed hash tables. This design principle, which revolves around mapping keys to their corresponding values, provides a highly scalable and flexible approach to data management. In this section, we will delve into the architecture and design considerations of key-value stores, uncovering the inner workings of these streamlined databases.

**Data Model**

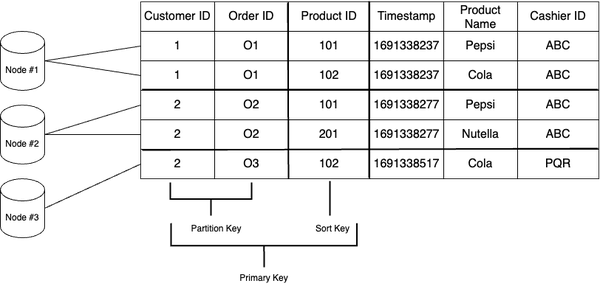
The fundamental concept of a key-value store revolves around a basic data structure - the key-value pair. The data is stored in tables, similar to RDBMS but the table abstraction is built over the key value store. Each key is associated with a corresponding value, forming a unit of data storage, called item. The keys are typically unique within the database and are used to access and retrieve their associated values. The values in a key-value store can be of various types, including strings, numbers, binary data, or even complex structures like JSON objects. Let’s go over the data model characteristics of key-value stores with respect to schemaless design and key implementation.

**No Fixed Schema or Indexes**

Unlike traditional relational databases, key-value stores do not enforce a fixed schema. This means that different key-value pairs can have varying structures, allowing for dynamic data models. So, some item records can have attributes that other records don’t have. Additionally, key-value stores typically lack complex indexing mechanisms, as they rely heavily on the efficiency of key-based lookups for data retrieval. Indexing, if present, is often limited to the keys themselves, optimizing the process of locating the desired values.

**Keys**

In a key-value store, the primary key, partition key, and sort key are fundamental concepts that help organize and retrieve data efficiently, as shown in [Figure 3-1](https://learning.oreilly.com/library/view/system-design-on/9781098146887/ch03.html#fig_1_keys_in_key_value_store). Let’s explore each of these concepts in more detail:



**Figure 3-1. Keys in Key Value Store**

*Primary Key*

The primary key is a unique identifier associated with each key-value pair in the key-value store. It serves as the primary means of accessing and retrieving data. The primary key provides a direct mapping to the corresponding value, allowing for fast retrieval operations. Typically, the primary key is a simple data element such as a string, integer, or a combination of multiple fields. By ensuring uniqueness, the primary key guarantees that each key-value pair is uniquely identifiable within the store.

*Partition Key*

The partition key is a subset of the primary key and is responsible for data distribution across multiple storage partitions in a distributed key-value store. It determines the storage location of the data within the store. The partition key is used to partition the data set into smaller, manageable subsets that can be distributed across different physical or virtual storage nodes. Each partition operates independently, allowing for horizontal scaling and improved performance. By selecting an appropriate partition key, data can be evenly distributed and evenly balanced across the partitions, avoiding hotspots and ensuring efficient storage and retrieval operations.

*Sort Key*

The sort key, also known as a range key, is an optional attribute used to order or sort the data within a partition. While the primary key uniquely identifies each key-value pair, the sort key allows for efficient range queries and data retrieval in a specific order. The sort key is particularly useful when you want to query a range of data based on a specific criterion, such as retrieving all records with timestamps within a certain range or retrieving data in alphabetical or numerical order. By organizing data based on the sort key, the key-value store can perform efficient range-based queries and optimize retrieval performance.

In summary, the primary key uniquely identifies each key-value pair in a key-value store. The partition key determines the data distribution across multiple storage partitions, enabling horizontal scalability and load balancing. The sort key, while optional, facilitates efficient sorting and range-based queries within each partition. Understanding and appropriately utilizing these key concepts in a key-value store helps optimize data storage, retrieval, and query performance.

**Data Access and Retrieval Operations**

In a key-value store, several operations are commonly used to manipulate data associated with specific keys. These operations include:

*GetItem*

The GetItem operation retrieves the attributes of an item associated with a given primary key. By specifying the primary key, the GetItem operation returns the set of attributes that describe the item. If no item exists for the provided key, the GetItem operation returns an empty result.

*PutItem*

The PutItem operation is used to insert an item into the store. If there is no existing item with the specified key, the operation creates a new item and associates it with the key. If an item already exists for the given key, the PutItem operation replaces the existing item with the new item.

*UpdateItem*

The UpdateItem operation is employed to modify an existing item. If an item with the specified key exists, the operation updates the attributes of that item. It allows adding new attributes, modifying existing ones, or removing attributes from the item. However, if no item exists for the given key, the UpdateItem operation adds a new item instead.

*DeleteItem*

The DeleteItem operation is used to remove an item from the key-value store. By providing the primary key of the item, the DeleteItem operation deletes the corresponding item from the store. If no item exists for the specified key, no action is taken.

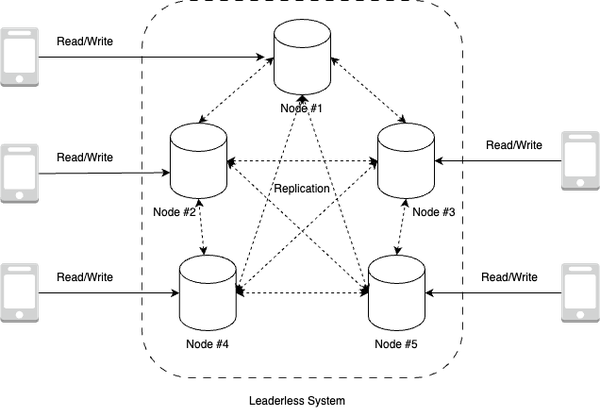
These operations form the basic set of functions used to interact with a key-value store, allowing for item insertion, update, deletion, and retrieval based on their associated keys. By utilizing these operations, developers can effectively manage and manipulate data within a key-value store.

**Scaling Key Value Stores**

Key-value stores are designed to handle massive volumes of data and support horizontal scalability. To achieve this, they often adopt a distributed architecture, where data is partitioned across multiple nodes or servers. Each node is responsible for storing a subset of the data, allowing the system to handle high loads and provide fault tolerance. Distributed key-value stores employ techniques such as leaderless replication using consistent hashing or partitioning algorithms to evenly distribute the data across the nodes. Let’s go through the leaderless replication technique in detail.

**Leaderless Replication**

Leaderless replication is a technique used in distributed systems, including key-value stores, to achieve high availability and fault tolerance without relying on a single designated leader node. Instead of having a dedicated leader responsible for coordinating read and write operations, all nodes in the system, as shown in [Figure 3-2](https://learning.oreilly.com/library/view/system-design-on/9781098146887/ch03.html#fig_2_leaderless_system_setup) are equal and can accept client requests independently.



**Figure 3-2. Leaderless System Setup**

In leaderless replication, data is typically divided into smaller partitions or shards, and each shard is replicated across multiple nodes in the system using peer to peer replication. This replication ensures data redundancy and fault tolerance. When a client wants to read or write data, it can send the request to any node in the system without needing to know which node is the leader. Each node has the capability to handle read and write operations, eliminating the bottleneck and single point of failure associated with a dedicated leader.

To ensure consistency, leaderless replication often employs techniques such as quorums and conflict resolution mechanisms. Quorums define the minimum number of successful responses required for an operation to be considered successful. For example, a quorum of N/2+1 nodes may be required to acknowledge a write operation to ensure that a majority agrees on the state change. Conflict resolution mechanisms such as vector clocks and Merkle trees help resolve conflicts that may arise when different nodes receive conflicting updates for the same data. Merkle trees assist in conflict resolution by efficiently identifying the specific differences between two versions of a dataset. By comparing the root hashes of these trees, one can pinpoint the exact branches where discrepancies occur, simplifying the process of identifying and reconciling conflicting data.

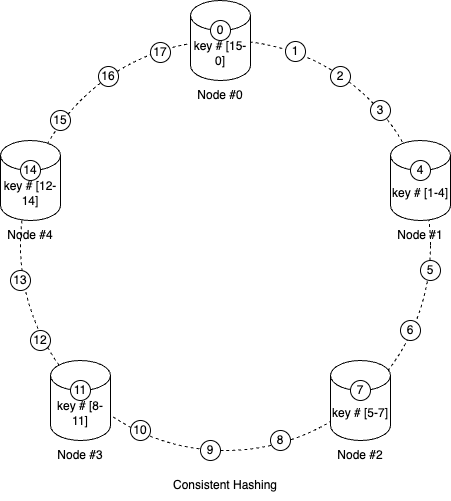
Leaderless replication provides high availability and fault tolerance because the system can continue operating even if some nodes become unavailable. Clients can send requests to any available node, and the system handles the replication and coordination transparently. However, ensuring strong consistency across the replicas can be more challenging in a leaderless replication model compared to systems with a designated leader. Hence, Amazon DynamoDB is based on leader-follower architecture, which we discussed in Chapter 1.

**Consistent Hashing**

Consistent hashing is a hashing technique commonly used in distributed systems, including key-value stores, to distribute data across a set of nodes while minimizing the impact of node additions or removals on the overall data distribution.

In traditional hashing, data is hashed to determine the node responsible for storing or serving it. However, this approach becomes problematic when nodes are added or removed from the system. In such cases, the distribution of data needs to be recalculated, resulting in significant data movement and potential disruptions.

Consistent hashing, as shown in [Figure 3-3](https://learning.oreilly.com/library/view/system-design-on/9781098146887/ch03.html#fig_3_consistent_hashing_implementation) addresses this issue by introducing a ring-like structure that represents the set of nodes in the system. Each node is assigned a position on the ring using a hash function. Data is also hashed to a position on the ring. The node whose position is the closest clockwise to the data’s position becomes responsible for storing or serving that data.



**Figure 3-3. Consistent Hashing Implementation**

The key advantage of consistent hashing is that when a node is added or removed, only a fraction of the data needs to be remapped to new nodes. Most of the data remains assigned to the same nodes as before, minimizing the data movement and disruption in the system. This makes consistent hashing highly scalable and efficient in distributed environments.

Furthermore, consistent hashing provides load balancing among nodes since data is evenly distributed across the ring, and nodes tend to have a similar number of data partitions assigned to them. It also allows for easy scaling by adding or removing nodes, as the redistribution of data is limited to the affected partitions.

In summary, consistent hashing is a technique that enables efficient and scalable data distribution in distributed systems. It minimizes the impact of node additions or removals by requiring only a fraction of the data to be remapped, ensuring high availability, load balancing, and easy scalability in key-value stores and other distributed systems.

**Availability in Key-Value Stores**

In a distributed key-value store, ensuring high availability is crucial to maintaining system responsiveness and reliability. Several mechanisms are commonly employed to achieve availability in such environments. Let’s explore some of these mechanisms:

*Optimistic Replication*

Optimistic replication is a technique used to provide availability in the presence of network partitions or temporary failures. In this approach, multiple replicas of data are maintained across different nodes in the system. When a write operation occurs, the changes are propagated asynchronously to the replicas. Instead of waiting for acknowledgments from all replicas before considering the write operation successful, the system assumes success and continues processing. This optimistic approach reduces latency and allows for continued availability even if some replicas are temporarily unavailable. However, it introduces the possibility of temporary inconsistencies between replicas, which are eventually resolved through background synchronization processes.

*Sloppy Quorum and Last Write Wins*

Sloppy quorum and Last Write Wins (LWW) are techniques used to achieve availability and eventual consistency in the face of network partitions or replica unavailability. In sloppy quorum, a subset of replicas, rather than the full set, is required to acknowledge a read or write operation. By allowing a relaxed quorum requirement, the system can continue operating even if some replicas are unavailable. However, this relaxation can result in temporary inconsistencies between replicas. To resolve conflicts in write operations, the Last Write Wins (LWW) strategy is often employed. In LWW, if multiple replicas receive conflicting updates for the same key, the update with the latest timestamp (according to a predefined ordering mechanism) takes precedence. This approach sacrifices strong consistency for availability, ensuring that the most recent write is eventually propagated to all replicas.

*Hinted Handoff*

Hinted Handoff is a mechanism used to handle temporary unavailability or network partitions in a distributed key-value store. When a write operation is performed on a replica that cannot immediately communicate with the primary replica, the write is temporarily stored in the form of a “hint.” These hints are later delivered to the appropriate replica once it becomes available again. This approach allows for continued availability of write operations, even in the presence of temporary replica failures. By leveraging hinted handoff, the system ensures that no data is lost and that all updates are eventually applied to the appropriate replicas.

These availability mechanisms, including optimistic replication, sloppy quorum, LWW, and hinted handoff, enable key-value stores to provide high availability in distributed environments. By balancing the trade-offs between availability and consistency, these mechanisms ensure that the system remains responsive and resilient, even in the face of network partitions, temporary failures, or replica unavailability.

**Advantages, Trade-offs and Considerations**

One of the primary advantages of key-value stores is their ability to deliver high performance and low-latency access to data. Since the data is accessed directly using a key, the retrieval process is highly efficient and typically involves minimal processing overhead. Key-value stores excel in use cases where rapid read and write operations are essential, such as caching, session management, and real-time data processing.

While key-value stores offer remarkable scalability and performance benefits, they do come with certain trade-offs. One of the major considerations is the limited querying capabilities. Key-value stores are optimized for simple key-based lookups and lack the advanced querying capabilities provided by relational databases. Moreover, transactions and complex operations involving multiple keys can be challenging to implement in a distributed key-value store.

Key-value stores find extensive use in various domains, including web applications, distributed systems, and caching layers. They excel in scenarios where fast and direct access to data is crucial, such as user session management, user profiles, product catalogs, and real-time analytics.

**Open Source Key Value Databases**

Open-source key-value databases provide developers with flexible and scalable solutions for managing data in a distributed and highly available manner. One popular open-source key-value store, Dynamo, has gained significant attention and adoption.

Dynamo is a highly available and scalable key-value store developed by Amazon. It was designed to handle the demanding requirements of Amazon’s shopping cart service. While Dynamo itself is not open-source, its principles have influenced the development of various open-source implementations such as Riak, Voldemort, and Dynomite.

Features of Dynamo include:

*Data Model*

Dynamo follows a simple key-value data model, where each item is uniquely identified by a key. The value associated with the key can be any binary data.

*Consistency Model*

Dynamo uses a tunable eventual consistency model, which means that updates may not be immediately reflected across all replicas.

*Query Language*

Dynamo does not provide a built-in query language. Instead, it relies on simple key-based operations for data access.

*Highly Available*

Dynamo prioritizes high availability, ensuring that data remains accessible even in the face of failures or network partitions.

*Horizontal Scalability*

Dynamo supports horizontal scaling by distributing data across multiple commodity hardware servers, allowing for increased storage capacity and improved performance with reduced compute cost.

**NOTE**

AWS offers Amazon DynamoDB, which is inspired by [the paper on Dynamo](http://www.cs.cornell.edu/courses/cs5414/2017fa/papers/dynamo.pdf) and its initial version, which we will cover in more detail in Chapter 10 - AWS Storage Services.

In conclusion, the architecture and design of key-value stores prioritize simplicity, scalability, and high-performance data access. By leveraging a straightforward key-value data model and distributed storage, these databases empower organizations to efficiently handle large-scale data management challenges while providing rapid and reliable data retrieval capabilities. Understanding the principles behind key-value stores equips data professionals with a valuable tool in their arsenal to tackle modern data-intensive applications.

Let’s go over the next type of non-relational database based on the document data model, storing and managing data in the form of documents.

**Document Databases**

Document databases are a type of non-relational database that are specifically designed for storing, retrieving, and managing semi-structured data in the form of documents. They provide a flexible and schema-less approach to data storage, making them ideal for applications with dynamic and evolving data structures. In this section, we will explore the architecture and design considerations of document stores, shedding light on their key features and benefits.

**Data Model**

At the core of a document store lies the document data model. Instead of organizing data into tables with predefined schemas, document stores store data as self-contained documents analogous to rows in relational databases, typically represented in formats such as JSON, BSON, or XML. These documents can have varying structures and can contain nested fields, arrays, and key-value pairs, allowing for hierarchical representation of data. The documents are organized into collections, which are analogous to tables in a relational database. The document model provides the ability to store and retrieve complex data structures as a single unit, making it well-suited for handling unstructured or semi-structured data. Let’s explore the key components of a document store schema and its architecture.

*Collections*

Collections in a document store are containers that hold related documents. They provide logical grouping of data, allowing you to organize and manage data based on its characteristics or purpose. For example, in an e-commerce application, you might have collections for products, orders, and customers. Each collection can have its own set of documents, each representing an individual entity or record.

*Documents*

Documents are the fundamental unit of data in a document store. They are analogous to rows in a relational database. A document is a JSON-like structure that stores data as key-value pairs. It represents a single entity or record and can contain nested structures and arrays. Documents within a collection can have different sets of fields, allowing for flexible data structures. This dynamic nature of document stores is particularly advantageous in scenarios where data evolves rapidly or when dealing with diverse and unpredictable data sources. For example, within a “products” collection, each document could have fields such as “name,” “price,” “description,” and so on.

*Operators*

Document stores provide a rich set of operators to perform various operations on documents. These operators allow you to query, update, and manipulate data within documents. Common operators include:

*Insert:*

Inserts a new document into a collection.

*Update:*

Modifies an existing document by updating or adding fields.

*Delete:*

Removes a document from a collection.

*Query:*

Retrieves documents from a collection based on specified criteria.

*Aggregation:*

Performs complex data aggregations and transformations on documents, such as grouping, sorting, and aggregating data.

*Projection*

Projection is a powerful feature in document stores that allows you to retrieve only the desired fields from a document. With projection, you can specify which fields to include or exclude in the result set. Indexing mechanisms, such as secondary indexes, are commonly used to optimize query performance, enabling fast access to the desired documents based on specific fields or criteria. This can significantly reduce network traffic and improve query performance, as only the required data is transferred from the database to the client.

In summary, document stores offer a flexible and schema-less approach to data storage and retrieval. Collections group related documents, while documents represent individual entities or records. Operators provide functionality for data manipulation, while projection allows for selective retrieval of fields. The distributed architecture of document stores enables high availability, fault tolerance, and scalability. With their rich features and flexible data model, document stores are well-suited for a wide range of use cases, from content management systems to real-time analytics applications.

**Availability in Document Stores**

The architecture of a document store typically involves a distributed system with multiple nodes. These nodes work together to provide high availability, fault tolerance, and scalability. Each node can handle read and write operations independently, allowing for parallel processing and improved performance. Data within a collection is often partitioned or sharded across multiple nodes to distribute the workload and enable horizontal scaling.

Document stores employ various mechanisms to achieve availability. Let’s explore three key components: replica sets, primary-secondary node clusters, and the heartbeat mechanism.

*Replica Sets*

A replica set is a group of nodes in a document store that contains multiple copies of the data. Each replica set consists of a primary node and one or more secondary nodes. The primary node is responsible for handling write operations and acting as the primary source of data. Secondary nodes replicate data from the primary node and serve as backups.

Replica sets provide fault tolerance by allowing automatic failover in the event of a primary node failure.

*Primary-Secondary Node Clusters*

In a primary-secondary node cluster architecture, the primary node accepts write operations and maintains the authoritative copy of the data. Secondary nodes replicate the data from the primary node and serve as read replicas. Clients can read from any of the secondary nodes, distributing the read workload and improving read performance.

The primary-secondary node cluster architecture ensures that read operations are scalable and can be handled by multiple nodes. In the event of a primary node failure diagnosed by heartbeat mechanism, one of the secondary nodes is promoted as the new primary, and the cluster continues to operate without data loss.

*Heartbeat Mechanism*

The heartbeat mechanism is responsible for monitoring the health and status of nodes, detecting failures, initiating failover processes, and promoting secondary nodes as new primaries when needed. It involves regular communication between nodes to detect failures and trigger appropriate actions. Nodes in a document store periodically exchange heartbeat messages to signal their availability. If a node fails to respond within a specified time period, it is considered unavailable, and the replica set or cluster takes necessary action to maintain availability.

Overall, these mechanisms provide fault tolerance, automatic failover, data redundancy, and efficient read scaling. By leveraging these features, document stores can maintain continuous access to data, withstand failures, and deliver consistent and reliable performance to applications and users.

**Advantages, Trade-offs and Considerations**

Document stores find extensive use in various application domains, including content management systems, e-commerce platforms, real-time analytics, and mobile app development. They excel in scenarios where flexibility in data modeling, dynamic schema evolution, and efficient querying of complex data structures are paramount. Document stores also facilitate the storage and retrieval of unstructured or semi-structured data, making them suitable for scenarios involving user-generated content, sensor data, log files, and social media feeds.

While document stores provide flexibility and scalability, there are trade-offs to consider. The flexibility of the schema-less design can sometimes lead to data consistency challenges, as the enforcement of data integrity constraints may be delegated to the application layer. Additionally, the lack of strong schema enforcement and complex joins can impact certain types of analytical or reporting queries that rely heavily on relationships and aggregations.

In summary, the architecture and design of document stores revolve around the document data model, flexible schema-less storage, distributed scalability, and powerful querying capabilities. By embracing the document-oriented approach, these databases empower organizations to handle semi-structured and evolving data with ease, enabling efficient storage, retrieval, and manipulation of complex data structures. Understanding the principles behind document stores equips data professionals with a versatile tool for managing diverse and dynamic data.

**Open-source Document Databases**

[MongoDB](https://www.mongodb.com/) is a widely adopted open-source document database known for its scalability, performance, and developer-friendly features. It uses a flexible JSON-like document model, allowing developers to store and retrieve data in a schema-less manner. MongoDB supports dynamic schemas, which means that each document in a collection can have its own unique structure.

Features of MongoDB include:

*Flexibility*

MongoDB’s schema-less design enables flexible data models and easy adaptation to evolving application requirements. It allows developers to handle varying and evolving data structures within the same collection.

*Rich Query Language*

It provides a powerful query language with support for complex queries, indexing, and aggregation. It allows developers to express a wide range of query operations and transformations.

*Multi-document Transactions*

Transactions in MongoDB adhere to the ACID properties, discussed in detail in Chapter 2. MongoDB supports multi-document transactions, enabling developers to work with multiple documents in a single transaction. This is particularly useful in scenarios where data in multiple collections needs to be updated together while maintaining data integrity.

*Data Consistency Models*

It supports various consistency models, allowing developers to choose the level of consistency needed for their applications.

*Horizontal Scalability*

It supports horizontal scaling through sharding, allowing data to be distributed across multiple nodes. This enables high performance and the ability to handle large data volumes and high traffic loads.

*Replication and High Availability*

Iit supports replica sets, which are self-healing clusters that provide data redundancy and automatic failover. Replica sets ensure high availability and fault tolerance by maintaining multiple copies of data.

In conclusion, MongoDB is a popular open-source document store with unique features and capabilities. It has a large and active community, providing extensive support, documentation, and a wide range of third-party integrations. MongoDB’s flexibility and community support make it suitable for various applications and deployment scenarios, including both on-premises and cloud.

**NOTE**

AWS offers Amazon DocumentDB, which is MongoDB compatible with support for powerful ad-hoc queries and comes with transaction support similar to Amazon DynamoDB, which we will cover in more detail in Chapter 10 - AWS Storage Services.

Let’s go over the next type of non-relational database based on columnar data model, organizing data into column families, allowing efficient storage and retrieval of large volumes of data.

**Columnar Databases**

Columnar databases or column-family databases, also known as Wide column stores, are a type of non-relational database that offer a unique architecture optimized for handling vast amounts of structured and semi-structured data. These databases excel at managing large-scale distributed systems, analytics, and use cases requiring fast read and write performance. In this section, we will explore the architecture and design considerations of wide column stores, unveiling their key features and advantages.

**Data Model**

Unlike traditional relational databases that organize data into rows, wide column stores employ a column-oriented data model with columns grouped into column families or column groups. In this model, data is stored and retrieved by columns rather than by rows. Each column represents a particular attribute or field, and data values belonging to that attribute are stored contiguously on disk. This design offers significant advantages in terms of query performance, as it allows for efficient compression, data skipping, and column-level operations like filtering and aggregation.

Columnar databases employ various compression techniques to improve storage efficiency and query performance. Since each column is stored separately, column-oriented storage allows for better compression ratios. Compression can be applied individually to each column based on its data characteristics, such as data type or redundancy. This not only reduces storage requirements but also improves data access speed by reducing disk I/O and memory footprint.

**Flexible Schema Design**

The schema design in a wide column store is flexible, enabling the addition or removal of columns without altering the entire dataset. This flexibility allows for easy adaptation to changing business requirements and evolving data models. Schema changes can be performed independently for each column family, providing greater agility in managing data structures.

**Keys**

Wide column stores typically use two types of keys: the partition key and the clustering key, which constitute the composite primary key.

*Partition Key*

The partition key is used to distribute data across the nodes in a cluster. It determines the physical location where data is stored. Data is partitioned based on the partition key, and each partition is stored on a separate node. Efficient selection of the partition key is crucial for even data distribution and optimal query performance.

*Clustering Key*

The clustering key is used to define the order of data within a partition. It allows for efficient sorting and range-based queries within a partition. The clustering key can consist of multiple columns, defining a hierarchical sorting order for the data.

Choosing the appropriate partition key and clustering key is a crucial aspect of designing a wide column store database. These keys determine how data is distributed, stored, and accessed within the database. Partition key should be chosen on a column with high cardinality, aiming for even data distribution, while clustering key should be chosen based on query access pattern of the application depending on how data is retrieved in specific order or pattern.

**Consistency Levels**

Wide column stores provide different levels of consistency to balance performance and data integrity. Wide column stores often offer tunable consistency, allowing developers to configure consistency levels on a per-operation basis. This enables the fine-tuning of consistency requirements for specific read and write operations.

Consistency levels in tunable consistency within a wide column store are based on the concept of quorum, which refers to the minimum number of replicas that need to participate in a read or write operation. The number of replicas required to form a quorum can vary depending on the desired consistency level. Let’s explore the consistency levels and their corresponding quorum requirements:

*Eventual Consistency*

Eventual consistency is the weakest level of consistency, allowing for the highest degree of availability and low latency. In wide column stores, eventual consistency means that replicas are asynchronously updated over time, resulting in the potential for temporary data inconsistencies. For eventual consistency, the quorum requirements can be defined as:

* Any: In this model, the operation is considered successful as soon as it is applied to at least one replica, without the requirement for acknowledgment or synchronization across all replicas. Over time, the updates are propagated and eventually converge to a consistent state across the system.

*Weak Consistency*

Weak consistency is a less common consistency level and provides a lower level of data integrity compared to strong consistency. In wide column stores, weak consistency allows for more relaxed requirements, prioritizing availability and low latency over data consistency. For weak consistency, the quorum requirements can be defined as:

* Quorum: In this model with a quorum-based approach, the consistency level is achieved when a majority of replicas (more than half) acknowledge the operation. This means that for a successful read or write operation, the data is considered consistent if it is read or written to at least one replica within the quorum.
* Local Quorum: In this model, a local quorum refers to a majority of replicas within the local data center or region. The operation is considered consistent if it is completed within the local quorum.

*Strong Consistency*

Strong consistency provides the highest level of data consistency but may come with increased latency and reduced availability. In wide column stores, strong consistency ensures that all replicas are synchronously updated and consistent before a response is returned. For strong consistency, the quorum requirements can be defined as:

* All: In this model, the operation must be acknowledged and applied to all replicas in the cluster before it is considered consistent. This means that for a read or write operation to be successful, it must be propagated to and acknowledged by every replica in the system.
* Each Quorum: In this model over multiple data centers or regions, each quorum refers to a majority of replicas within each data center or region. The operation is considered consistent when it is completed within each quorum.

It’s important to note that the specific consistency levels and their corresponding quorum requirements may vary between different wide column store databases. The above examples provide a general understanding of how consistency levels are defined based on quorum in a tunable consistency model within a wide column store. When selecting a consistency level, it’s crucial to consider the trade-offs between consistency, availability, and performance based on the specific requirements of your application and use case.

In conclusion, wide column stores offer a flexible data model, allowing for dynamic schema evolution and efficient storage of structured and semi-structured data. The schema design is adaptable, and keys (partition and clustering) play a crucial role in data distribution and query optimization. Consistency levels provide options to balance performance and data integrity. Understanding the wide column store data model, schema design, keys, and consistency levels is essential for effectively utilizing wide column stores in various applications and use cases.

**Columnar Store Architecture**

This section discusses columnar store architecture, which may vary from one implementation to other but generally, the core components remain the same. Let’s explore the key components of data storage in columnar storage and how they contribute to efficient data management.

*Commit Log*

The commit log is a write-ahead log that records all write operations performed on the database. It serves as a durable and sequential log of changes made to the data. Whenever a write operation occurs, it is first written to the commit log for durability and fault tolerance. The commit log ensures that no data is lost in the event of a system failure or crash.

*Memtable*

The memtable is an in-memory data structure that stores recent write operations before they are flushed to disk. It acts as a write buffer, temporarily holding the data in memory before persisting it to disk in an efficient manner. The memtable is typically structured as an ordered hash table or skip list, allowing for fast write operations. As the memtable fills up, it is periodically flushed to disk, creating SSTables.

*SSTables (Sorted String Tables)*

SSTables are the on-disk data structure in columnar storage. They are immutable and sorted by key, enabling efficient range queries and compression. Each SSTable contains a set of sorted key-value pairs, where the keys correspond to the row keys and the values represent the columnar data for each key. SSTables are optimized for fast read operations and can be compacted to improve performance and storage efficiency. Columnar storage leverage Bloom filters to ascertain if an SSTable contains data related to a specific partition. Bloom filters are used for index scans but not for range scans. Bloom filters function as probabilistic sets ​​to quickly test whether an element is a member of the set, offering a memory-accuracy trade-off. It’s particularly efficient in cases where false positives are acceptable but false negatives are not desired.

*Compaction Strategies*

Compaction is the process of merging and compacting multiple SSTables to improve read performance and manage disk space. Various compaction strategies are employed to balance read and write performance, as well as disk space utilization:

*Size Tiered Compaction:*

This strategy groups SSTables into levels based on their size. As SSTables in a level reach a certain threshold, they are merged into a new SSTable in the next level. It reduces the number of SSTables and simplifies data access at the cost of some write amplification.

*Leveled Compaction:*

In this strategy, SSTables are organized into multiple levels, with each level containing SSTables of roughly equal size. SSTables are compacted within each level, and as they progress to higher levels, they are merged with larger SSTables. Leveled compaction offers more balanced read and write performance at the expense of higher disk space overhead.

*TimeWindow Compaction:*

This strategy is specifically designed for time-series data, where data is organized based on a time window. It allows for efficient expiration of old data by dropping or merging SSTables based on time-based criteria.

*Tombstones for Soft Delete*

Tombstones are special markers used for soft deletes in columnar storage. When a row or column is deleted, a tombstone is created to indicate the deletion. Tombstones ensure that deleted data is properly handled during compaction and read operations, maintaining data consistency across SSTables. During compaction, tombstones are used to identify and remove deleted or expired data.

Excessive tombstones in columnar storage can lead to significant increase in garbage collection (GC) pauses during compaction, as columnar stores need to process and eliminate these tombstones, slowing down data access and increasing latency. Also, large numbers of tombstones consume storage space and querying data in the presence of numerous tombstones can result in slower query performance as the columnar database has to shift through these markers, potentially causing unnecessary overhead and delays in retrieving relevant data. Effective tombstone management is crucial to maintaining columnar store’s performance and responsiveness.

In summary, data storage in columnar storage involves various components such as the commit log, memtable, SSTables, compaction strategies, and tombstones. These components work together to ensure durability, efficient write buffering, disk storage optimization, and data consistency. Understanding these elements is crucial for effectively managing and leveraging the benefits of columnar storage in data-intensive applications.

**Advantages, Trade-offs and Considerations**

Wide column stores find widespread use in various domains, particularly in analytics, big data processing, and time-series data. They are well-suited for applications that involve complex queries, ad hoc analysis, data warehousing, and high-speed data ingestion. Use cases include log analysis, financial analytics, real-time reporting, IoT data processing, and customer behavior analysis.

While wide column stores offer significant advantages in scalability and query performance, they come with trade-offs. Due to their distributed nature and complex data organization, they often require more advanced data modeling and query optimization compared to traditional relational databases. Wide column stores may not be as suitable for scenarios that heavily rely on complex joins across multiple tables.

In conclusion, the architecture and design of wide column stores revolve around the column-oriented data model, distributed scalability, and efficient analytics capabilities. By leveraging the benefits of columnar storage and flexible schema design, these databases enable organizations to efficiently store, retrieve, and analyze large volumes of structured as well as semi-structured data.

**Open Source Columnar Databases**

[Apache Cassandra](https://cassandra.apache.org/_/index.html) is a highly scalable and distributed open-source columnar database known for its ability to handle massive amounts of structured and semi-structured data across multiple commodity servers. It offers a robust architecture and a range of features that make it suitable for high-performance and fault-tolerant applications. Let’s delve into Apache Cassandra and explore its key features.

Features of Cassandra include:

*Distributed Query Language (CQL)*

Cassandra utilizes CQL, a query language similar to SQL, to interact with the database. CQL provides a familiar syntax for developers and offers features like data definition, data manipulation, and querying capabilities. It simplifies data access and supports complex querying, filtering, and sorting operations.

*Distributed and Decentralized Architecture*

Cassandra follows a distributed architecture model, employing a peer-to-peer approach. It organizes data across a cluster of nodes, allowing for easy scalability and fault tolerance. Each node in the leaderless cluster can perform read and write operations independently, resulting in a distributed and highly available database system.

*Linear Scalability*

Cassandra’s architecture enables linear scalability, meaning it can handle increasing workloads by simply adding more nodes to the cluster. This scalability allows for seamless growth as data volumes and user demand increase, making it a suitable choice for applications with rapidly growing datasets.

*High Availability and Fault Tolerance*

Cassandra provides built-in high availability and fault tolerance mechanisms. It uses peer-to-peer replication across nodes, ensuring data redundancy and preventing data loss in the event of node failures. Cassandra automatically replicates data across multiple nodes based on the replication factor, guaranteeing data availability and durability.

*Tunable Consistency*

Cassandra offers tunable consistency, allowing developers to define the level of consistency required for each read and write operation. This flexibility enables balancing performance and data consistency based on application-specific needs. Developers can configure consistency levels from strong consistency to eventual consistency.

*Flexible Data Replication and Data Centers*

Cassandra allows the replication of data across multiple data centers or geographic regions, providing geographical redundancy and disaster recovery capabilities. It supports multiple replication strategies, including network topology-aware data replication, ensuring data availability and low-latency access in distributed environments.

In summary, Apache Cassandra is an open-source columnar database that offers scalability, high availability, fault tolerance, tunable consistency, and support for distributed and decentralized architectures. With its columnar data model, distributed query language (CQL), time-series data capabilities, and active community support, Cassandra empowers developers to build robust and scalable applications that can handle massive amounts of data across distributed environments.

**NOTE**

AWS offers Amazon Keyspaces, which is Apache Cassandra compatible, highly scalable, available, and managed wide-column database service offered as a serverless solution, which we will cover in more detail in Chapter 10 - AWS Storage Services.

Let’s go over the next type of non-relational database based on a graph data model, well-suited for applications that require complex querying and analysis of interconnected data..

**Graph Databases**

Graph databases are a specialized type of non-relational database designed to handle highly interconnected data and complex relationships. They excel at storing, querying, and traversing graph-like structures, making them ideal for scenarios involving social networks, recommendation systems, fraud detection, and knowledge graphs. In this section, we will explore the architecture and design considerations of graph stores, uncovering their key features and advantages.

**Data Model**

At the heart of a graph store lies the graph data model, which represents data as a collection of interconnected nodes (vertices) and relationships (edges). Each node typically corresponds to an entity or an object, while the relationships represent the connections between nodes. Graph databases store these entities, relationships, and associated properties, allowing for efficient traversal and querying of the graph structure. The graph data model offers a natural representation of complex and interconnected data, enabling rich and expressive data modeling.

Unlike traditional databases that focus on data entities and their attributes, graph stores prioritize the relationships between entities. Relationships in graph databases can have properties and can be directed or undirected, allowing for various types of connections. The relationship-centric design of graph stores enables efficient navigation of the graph, facilitating queries that traverse nodes and relationships to uncover patterns, paths, and insights within the data.

**Data Access and Retrieval**

Graph stores provide powerful query languages or APIs specifically designed for traversing and querying graph structures. These languages, such as Cypher (used in Neo4j) or Gremlin (used in Apache TinkerPop), allow users to express complex graph patterns, perform graph traversals, and filter and aggregate data based on relationships and properties. The query languages provide a declarative and expressive syntax that simplifies the process of querying and exploring the graph data.

Graph stores employ various indexing techniques to optimize graph traversal and query performance. They typically use indexes on nodes, relationships, or properties to speed up the lookup of specific elements within the graph. Additionally, graph databases often utilize caching mechanisms to store frequently accessed graph elements in memory, further improving query response times. These indexing and caching strategies enable graph stores to efficiently handle complex graph traversals and pattern matching queries.

**Advantages, Trade-offs and Considerations**

Graph stores are particularly well-suited for applications that involve complex relationships, network analysis, and recommendation systems. They find extensive use in social networking platforms, fraud detection systems, recommendation engines, knowledge graphs, and data lineage analysis. Graph databases excel in scenarios where understanding and analyzing the relationships between entities are of primary importance.

While graph stores offer powerful graph traversal and querying capabilities, they may face challenges when dealing with massive datasets and complex graph patterns. The performance of graph queries heavily depends on the size and structure of the graph, and certain complex queries may require additional optimization techniques. Additionally, graph stores may not be the best choice for scenarios that primarily involve simple tabular data or scenarios that require strong transactional capabilities.

In summary, the architecture and design of graph stores revolve around the graph data model, relationship-centric design, efficient graph traversal and querying, and scalable distributed architectures. By leveraging the power of graph structures and relationships, these databases enable organizations to store, navigate, and uncover valuable insights within highly interconnected datasets.

**Open-source Graph Databases**

[Neo4j](https://neo4j.com/) is a leading open-source graph database known for its powerful graph processing capabilities and intuitive query language. Neo4j is designed to efficiently store, manage, and traverse highly connected data, making it ideal for applications that heavily rely on complex relationships and interconnections.

Features of Neo4j include:

*Graph Data Model*

Neo4j is built around the property graph data model, which consists of nodes, relationships, and properties. Nodes represent entities or objects, relationships define connections between nodes, and properties store key-value pairs associated with nodes and relationships. This flexible and expressive data model allows developers to represent and explore complex relationships easily.

*Graph Query Language*

Neo4j uses Cypher, a powerful and intuitive query language specifically designed for graph databases. Cypher allows developers to express complex graph patterns and perform advanced graph traversals using a familiar and human-readable syntax. It provides rich querying capabilities, including filtering, aggregations, and pattern matching, making it easier to extract meaningful insights from highly connected data.

*Scalability and Performance*

Neo4j offers excellent scalability and performance for graph data processing. It supports horizontal scaling by distributing the graph across multiple machines in a cluster, enabling high throughput and accommodating large-scale deployments. Neo4j’s query optimizer efficiently executes queries, leveraging index structures and caching mechanisms to ensure optimal performance even on massive graphs.

*ACID Compliance*

Neo4j provides strong ACID (Atomicity, Consistency, Isolation, Durability) guarantees to ensure data integrity and reliability. It maintains strict transactional consistency, allowing multiple operations to be grouped into atomic units of work. This ensures that changes made to the graph are durable and consistent, even in the presence of concurrent operations.

*Native Graph Processing*

Neo4j is designed from the ground up as a native graph database, allowing it to leverage the power of graph processing algorithms. It provides a wide range of graph-specific operations, such as shortest path calculations, centrality measurements, community detection, and graph analytics. These built-in capabilities enable developers to perform complex graph analysis and traverse relationships efficiently.

*Data Visualization*

Neo4j provides visualization tools that allow developers to explore and visualize graph data. These tools help understand the structure of the graph, identify patterns, and gain insights into complex relationships. Visualization enhances the intuitive understanding of the data model and aids in decision-making processes.

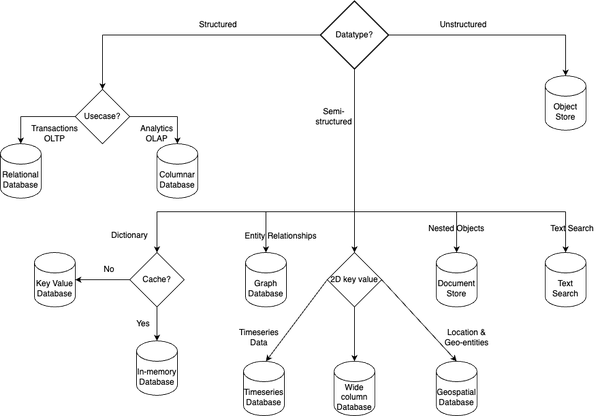
With its graph data model, powerful Cypher query language, scalability, performance, ACID compliance, native graph processing capabilities, vibrant community, and visualization tools, Neo4j empowers developers to build sophisticated applications that leverage the power of relationships and graph data. Whether it’s social networks, recommendation systems, fraud detection, or knowledge graphs, Neo4j provides a robust foundation for graph-based applications.

**NOTE**

AWS offers fully managed graph database service, Amazon Neptune, which we will cover in more detail in Chapter 10 - AWS Storage Services.

**Conclusion**

As we conclude our exploration of non-relational databases, it’s evident that the landscape is rich with diversity. From key-value stores optimized for rapid operations, to columnar stores tailored for analytical prowess, and document and graph databases catering to flexible data structures and intricate relationships, each type offers a unique set of capabilities. We did not cover other database types like timeseries (ex. InfluxDB), in-memory (ex. Redis),  text search (ex.  ElasticSearch), vector (Milvus), geospatial, ledger databases, but the choices are immense. The choice of database type, as shown in [Figure 3-4](https://learning.oreilly.com/library/view/system-design-on/9781098146887/ch03.html#fig_4_database_selection_decision_flowchart) should be guided by a deep understanding of your application’s requirements, workload characteristics, and growth expectations.



**Figure 3-4. Database selection decision flowchart**

We’ve taken a journey through different kinds of databases in Chapter 2 and 3, each with its own special abilities. An application may require a relational database or few non-relational database types along with it to support the specific use case. Think of them like different tools in a toolbox, ready to help us manage data in exciting ways.

Suppose, you have to put a nail in the wall and you have a hammer, wrench, screw-driver and other tools. Sure, using multiple tools means you have to learn how to handle them differently, but that’s okay. Each tool has its job, and it’s important to use the right one (the hammer in this case) for the task at hand. So, remember, always pick the right tool for the job, instead of trying to make one tool do everything or use all the tools – because it just doesn’t work that way!

Let’s relook at the comparison between relational and non-relational databases in Table 3-1 to refresh your knowledge.

|  |  |  |
| --- | --- | --- |
| Property | Relational Store | Non-relational Store |
| Data Model | Follows a strict schema. Data is stored in tables with well defined structure and adding columns to a defined table can be difficult. | Are schema-less. Data are stored in tables with well defined structure Data can be structured as key-value pairs, JSON and adding columns to a defined table can be like documents, or any of the other various other nosql designs like graph, wide column etc. |
| Hierarchical Storage | These databases are not suited for hierarchical data storage. | These databases are best suited for hierarchical or interconnected data storage. |
| Scalability | SQL databases are better suited for vertical scaling. | NoSQL databases are horizontally scalable. And as such can employ techniques like sharding etc to facilitate handling of huge amounts of data. |
| Joins | Suitable for queries that require a lot of complex joins across multiple tables. | Joins are possible but involve a lot of computational overhead to perform especially on a distributed database. |
| CAP Theorem | SQL transactions always comply with ACID properties (Strong Consistency). | NoSQL databases generally follow BASE properties (Eventual Consistency). |
| Table 3-1. Relational vs Non-relational Store | | |

As we conclude this chapter on non-relational databases, we hope you feel equipped with the knowledge and know-hows of databases, along with a basic understanding of when to choose which database. The next chapter will embark you on an exploration of caching policies and strategies, reducing the time required to access frequently accessed data, even from datastores.