UNIT-2

Distribution Models: Single Server, Sharding, Master-Slave Replication, Peer-to-Peer Replication, Combining Sharding and Replication, The CAP Theorem.

Key-Value Databases: What Is a Key-Value Store, Key-Value Store Features, Consistency, Transactions, Query Features, Suitable Use Cases, When Not to Use.

Distribution Models:-

- Distribution models refer to the ways in which data is distributed and stored across multiple nodes and clusters in a distributed computing environment.
- There are several distribution models that are commonly used in distributed systems, including the following:
    1. Single Server Distribution model
    2. Sharding Distribution model
    3. Master-Slave Replication Distribution model
    4. Peer-to-Peer Replication Distribution model
    5. Combining Sharding and Replication

1. Single Server Distribution model:

- The single server distribution model, also known as the centralized model, is a distribution model in which all data is stored on a single server or database.
- In this model, there is a single point of control for all data, and all requests for data are processed by the same server.
- The single server model is often used in small-scale applications or systems where the volume of data is relatively low and the performance requirements are not very demanding.
- This model is also commonly used in traditional relational databases, such as MySQL or Oracle, where data is stored in tables and accessed using SQL queries.

Advantages:

Simplicity: The single server model is simple and easy to manage since all data is stored on a single server. This makes it easy to ensure data consistency and integrity since there is only one copy of the data.

Cost-effective: The single server model is cost-effective since only one server is needed to store and manage all data.

Easy to back up: Since all data is stored on a single server, it is easy to back up the data and restore it in case of a disaster or failure.

Efficient for small-scale applications: The single server model is efficient for small-scale applications or systems with low data volumes and low performance requirements.

Easy to implement: The single server model is easy to implement since it does not require any complex setup or configuration.

Disadvantages:

Limited scalability: The single server model is not well-suited for large-scale applications or systems with high data volumes or demanding performance requirements. As data volume increases, the server may become overwhelmed and unable to handle the load, leading to performance issues and potential downtime.

Single point of failure: The single server model is less fault-tolerant than other distribution models since a failure of the single server could result in the loss of all data. This makes it critical to implement backup and disaster recovery measures to prevent data loss.

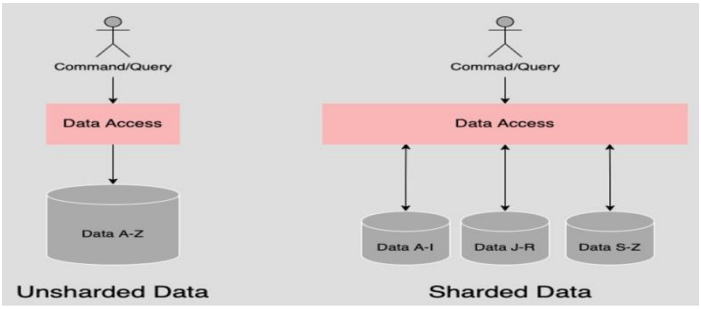
Limited redundancy: In the single server model, data is stored on a single server, which means there is limited redundancy. If the server fails, data recovery can be difficult and may result in significant downtime.

Performance bottlenecks: As data volume increases, the server may become a bottleneck, leading to decreased performance. This can be mitigated by implementing performance tuning measures, but it may not be sufficient for large-scale applications.

Limited geographical distribution: In the single server model, data is stored on a single server, which means that it can only be accessed from a single location. This can be a disadvantage for organizations with geographically distributed teams or customers.

2. Sharding Distribution model:

- Sharding is a type of data distribution model used in NoSQL databases to horizontally partition data across multiple servers or nodes.
- In this model, a large dataset is split into smaller subsets or shards, which are distributed across multiple servers.
- Each server stores a subset of the data, and requests for data are distributed across multiple servers, allowing for faster processing and improved scalability.



Advantages:

Scalability: The sharding distribution model allows for easy scaling of the database by adding additional servers or nodes to the system. As the data volume grows, new shards can be created, and additional servers can be added to distribute the load.

High availability: The sharding distribution model offers high availability since the data is distributed across multiple servers. If one server fails, the other servers can continue to handle requests, and data can be recovered from other shards.

Improved performance: The sharding distribution model offers improved performance since requests can be distributed across multiple servers, allowing for faster processing.

Efficient use of resources: The sharding distribution model allows for efficient use of resources since each server only stores a subset of the data, reducing the amount of storage and memory required on each server.

Geographic distribution: The sharding distribution model allows for geographic distribution of data, making it suitable for applications that require data to be stored in multiple locations.

Disadvantages:

Complex data management: Sharding requires complex data management, including partitioning the data, distributing the data across multiple servers, and ensuring data consistency and integrity.

Difficult data migration: Moving data between shards can be difficult and time-consuming, making it challenging to scale the system.

Increased complexity: Sharding adds complexity to the system architecture, making it more difficult to manage and troubleshoot.

Note- Overall, the sharding distribution model is a good choice for applications with high data volumes and performance requirements. Its scalability, high availability, and improved performance make it a popular choice for large-scale applications.

3. Master-Slave Replication Distribution model

- Master-Slave Replication is a type of data distribution model used in NoSQL databases where one node acts as the master node, and other nodes act as slave nodes.
- In this model, the master node receives all data writes and updates, and the slave nodes replicate the data from the master node.
- This model is used to improve the availability and scalability of the database system.

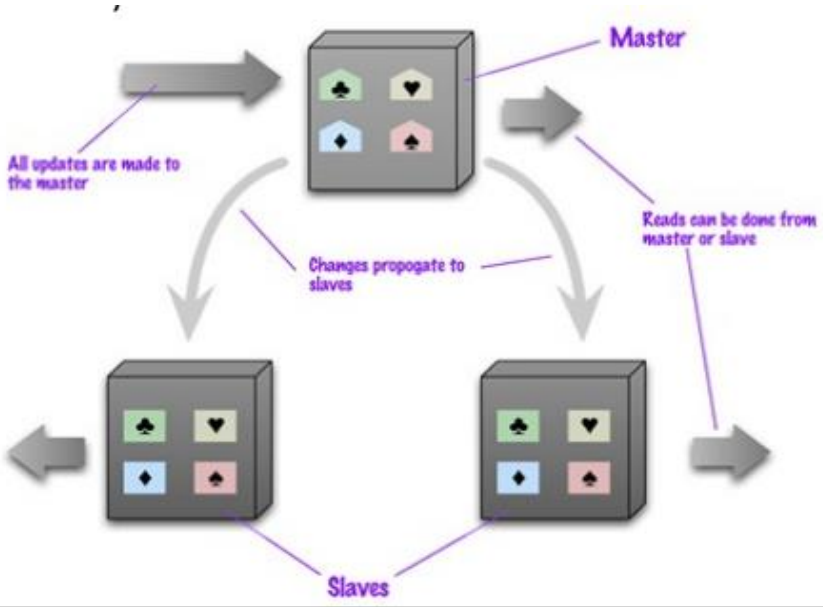


Figure: Data is replicated from master to slaves. The master services all writes; reads may come from either master or slaves.

Advantages:

Improved availability: The Master-Slave Replication model improves the availability of the database by providing redundancy. If the master node fails, the slave nodes can take over and continue to provide service.

Scalability: The Master-Slave Replication model provides scalability by allowing additional slave nodes to be added to the system. As data volume increases, additional slave nodes can be added to the system to distribute the load.

Faster read performance: The Master-Slave Replication model can improve read performance since slave nodes can handle read requests, allowing the master node to focus on write requests.

Geographic distribution: The Master-Slave Replication model can be used for geographic distribution of data, making it suitable for applications that require data to be stored in multiple locations.

Disadvantages:

Single point of failure: The Master-Slave Replication model has a single point of failure since the master node receives all data writes and updates. If the master node fails, the entire system may fail.

Increased complexity: The Master-Slave Replication model adds complexity to the system architecture, making it more difficult to manage and troubleshoot.

Potential data consistency issues: The Master-Slave Replication model can have potential data consistency issues if updates are made to the slave nodes before they are replicated to the master node.

4. Peer-to-Peer Replication Distribution model

- Master-slave replication helps with read scalability but doesn't help with scalability of writes. It provides resilience against failure of a slave, but not of a master. Essentially, the master is still a bottleneck and a single point of failure.
- Peer-to-Peer Replication is a type of data distribution model used in NoSQL databases where all nodes act as both a master and a slave, communicating with each other to distribute data updates.
- In this model, each node stores a portion of the data and replicates it to other nodes in the cluster.

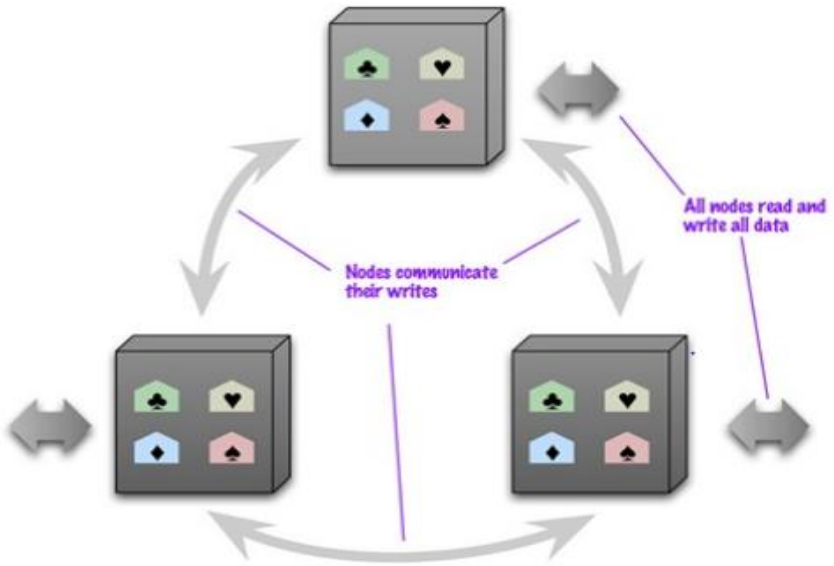


Figure: Peer-to-peer replication has all nodes applying reads and writes to all the data.

Advantages:

Improved scalability: The Peer-to-Peer Replication model provides scalability by allowing additional nodes to be added to the system. As data volume increases, additional nodes can be added to the system to distribute the load.

Improved fault tolerance: The Peer-to-Peer Replication model improves fault tolerance by removing the single point of failure that exists in the Master-Slave Replication model. If a node fails, the other nodes can continue to provide service.

Improved read performance: The Peer-to-Peer Replication model can improve read performance since each node can handle read requests, allowing data to be distributed across the cluster.

Geographic distribution: The Peer-to-Peer Replication model can be used for geographic distribution of data, making it suitable for applications that require data to be stored in multiple locations.

Disadvantages:

Increased complexity: The Peer-to-Peer Replication model adds complexity to the system architecture, making it more difficult to manage and troubleshoot.

Potential data consistency issues: The Peer-to-Peer Replication model can have potential data consistency issues if updates are made to different nodes at the same time, leading to conflicts.

Increased network traffic: The Peer-to-Peer Replication model can result in increased network traffic since data is replicated between all nodes.

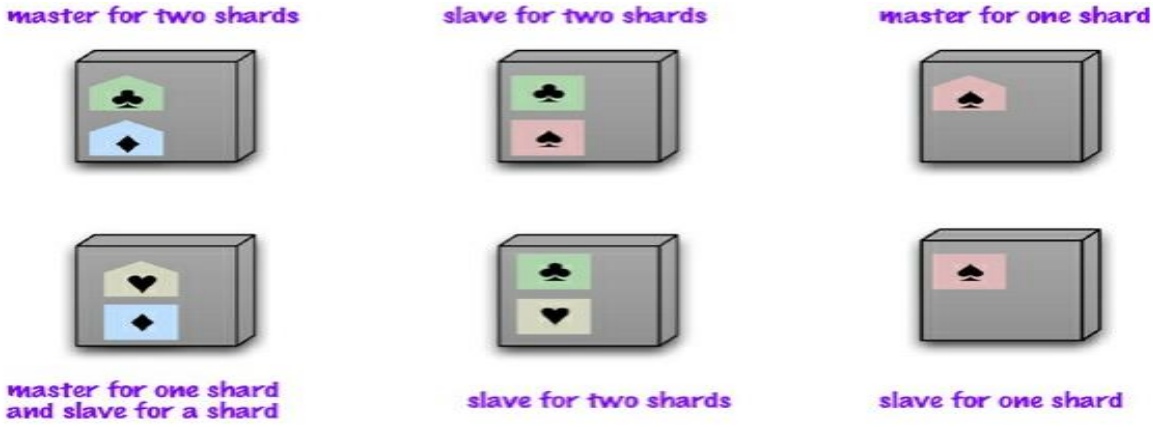
Note- Overall, the Peer-to-Peer Replication Distribution model is a good choice for applications that require high scalability, fault tolerance, and read-heavy workloads. Its distributed nature makes it suitable for large-scale applications and can provide improved performance and availability. However, the increased complexity and potential data consistency issues must be carefully managed.

5. Combining Sharding and Replication:

- Combining Sharding and Replication is a data distribution model used in NoSQL databases that combines the benefits of both Sharding and Replication.
- In this model, data is partitioned into smaller chunks using sharding and then replicated across multiple nodes to ensure high availability and fault tolerance.
- **Sharding** splits the data into smaller partitions called shards and distributes them across multiple nodes in the cluster. Each node is responsible for storing and processing a subset of the data, reducing the workload on each individual node and improving scalability.
- **Replication,** on the other hand, ensures that each shard is replicated across multiple nodes to provide fault tolerance and high availability. Each replica of a shard is stored on a different node, so if one node fails, the data is still available on the other nodes.

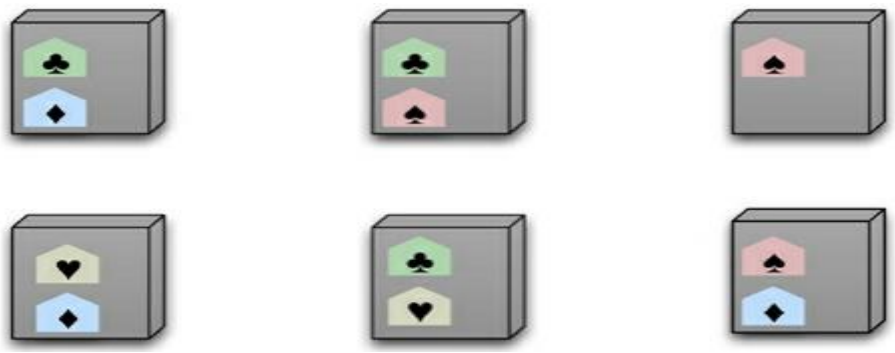
Using master-slave replication together with sharding

If we use both master-slave replication and sharding this means that we have multiple masters, but each data item only has a single master. Depending on your configuration, you may choose a node to be a master for some data and slaves for others, or you may dedicate nodes for master or slave duties.



Using peer-to-peer replication together with sharding

Using peer-to-peer replication and sharding is a common strategy for column-family databases. In a scenario like this you might have tens or hundreds of nodes in a cluster with data sharded over them. A good starting point for peer-to-peer replication is to have a replication factor of 3, so each shard is present on three nodes. Should a node fail, then the shards on that node will be built on the other nodes.



Advantages:

Improved scalability: Sharding enables horizontal scaling by distributing data across multiple nodes, while Replication ensures that data is available on multiple nodes, improving the scalability and performance of the system.

Improved fault tolerance: Replication provides fault tolerance by ensuring that each shard is available on multiple nodes, so if one node fails, the data is still available on other nodes.

Improved read performance: Sharding enables faster reads by distributing the data across multiple nodes, and Replication ensures that each shard is available on multiple nodes, improving read performance.

Geographic distribution: Combining Sharding and Replication allows data to be stored in multiple locations, making it suitable for applications that require geographic distribution of data.

Disadvantages:

Increased complexity: Combining Sharding and Replication adds complexity to the system architecture, making it more difficult to manage and troubleshoot.

Increased network traffic: The replication of data across multiple nodes can result in increased network traffic, which can affect system performance.

Data consistency issues: When updates are made to different replicas of a shard, data consistency issues may arise, leading to conflicts that must be resolved.

The CAP Theorem:

The CAP theorem is a fundamental concept in distributed computing that describes the trade-offs that must be made when designing a distributed system. It was first proposed by computer scientist Eric Brewer in 2000, and it has since become a widely accepted principle in the field.

The CAP theorem states that it is impossible for a distributed system to simultaneously provide all three of the following properties: Consistency, Availability, and Partition tolerance.

Consistency: Refers to the guarantee that all nodes in the system will return the same result for a given operation, regardless of which node is queried. In other words, all nodes in the system are in sync with each other and provide a consistent view of the data.

Availability: Refers to the guarantee that every request made to the system will receive a response, without any guarantee that it contains the most recent version of the data. In other words, the system is available to process requests even in the event of node failures.

Partition tolerance: Refers to the guarantee that the system will continue to function despite any number of messages being dropped or delayed by the network between nodes in the system. In other words, the system can continue to operate even if some nodes cannot communicate with each other.

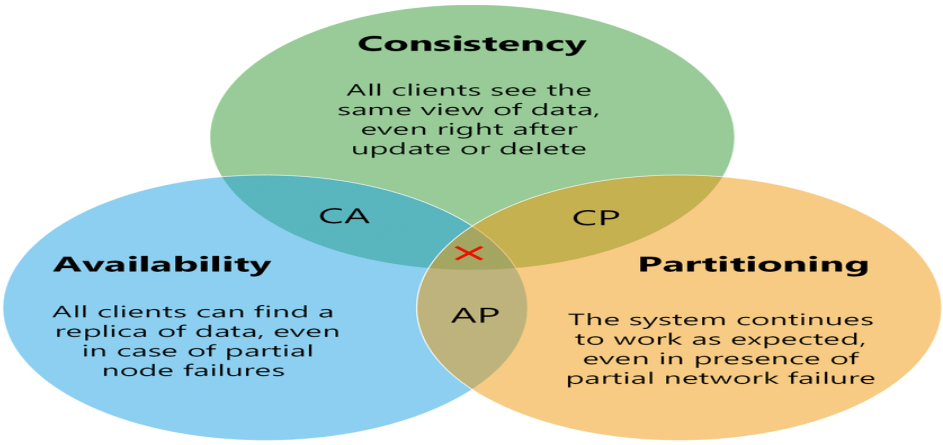
The CAP theorem states that in a distributed system, it is only possible to achieve two of the three properties mentioned above.

This means that a system can either be:

Consistent and partition tolerant, but not highly available. In this case, the system will be able to guarantee consistency and handle network partitions, but it may not always be able to provide a response to all requests, especially during network outages.

Available and partition tolerant, but not fully consistent. In this case, the system can always provide a response to requests, even if they are not always the most up-to-date, and can tolerate network partitions, but may not provide a consistent view of the data across all nodes.

Consistent and available, but not partition tolerant. In this case, the system can always provide a consistent view of the data and handle all requests, but it will not be able to tolerate network partitions, making it vulnerable to data loss or unavailability during network failures.

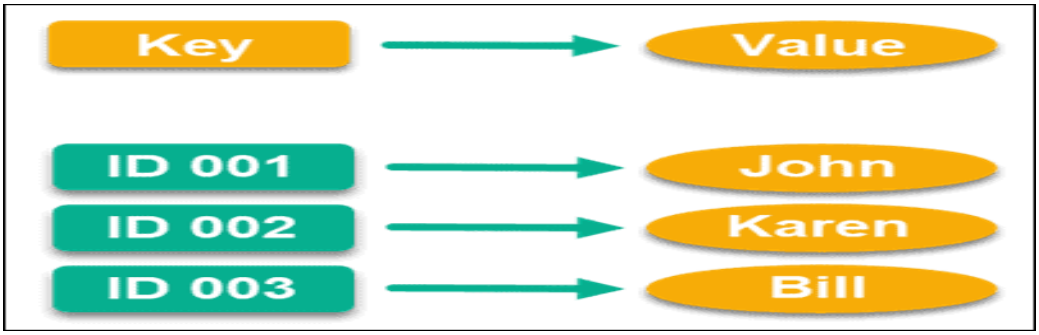


What Is a Key-Value Store:

- The key-value databases are designed for storing, retrieving, and managing a data structure known as a hash table (or dictionaries).
- A key-value store is a simple hash table, primarily used when all access to the database is via primary key.
- A key-value store is a type of NoSQL database which uses a key-value method to store data Each piece of data is associated with a unique key that acts as an identifier for that data.
- When a user wants to retrieve a piece of data, they use the associated key to look up the value stored in the database.
- key-value stores are highly flexible and can be used to store a wide variety of data types, from simple key-value pairs to more complex objects and data structures.
- Some key-value stores also support advanced data structures such as lists, sets, and maps, making them more versatile than traditional key-value databases.
- Some of the most popular key-value stores include **Redis, and Riak.**

Oracle	Riak
database instance	Riak cluster
table	bucket
row	key-value
rowid	key

UNIT-2



Key-Value Store Features

- While using any NoSQL data stores, there is an inevitable need to understand how the features compare to the standard RDBMS data stores that we are so used to.

The main difference between a key-value store and a relational database (RDBMS) lies in their data models and how they handle data. Here are some of the key differences:

	key-value store	RDBMS
Data model	Key-value stores have a simple data model where each value is associated with a unique key.	RDBMS use a table-based data model where data is organized into tables with defined relationships between them.
Schema	key-value stores are schema less, which means that the structure of the data is not defined in advance.	RDBMS require a pre-defined schema that specifies the structure of the data, including the table definitions, column names, and data types
Querying	Key-value stores, on the other hand, usually offer limited querying capabilities, typically allowing only for simple queries based on the key.	RDBMS use a SQL-based querying language that allows for complex queries to be performed on the data.
Scalability	Key-value stores are designed to scale horizontally by adding more nodes to a cluster.	RDBMS can be scaled vertically by adding more resources (such as CPU, memory, or storage) to a single server
Performance	Key-value stores are typically faster than RDBMS for simple queries and data lookups because they have a simpler data model and fewer dependencies.	RDBMS can be faster for complex queries involving multiple tables and relationships.
Consistency	Key-value stores, typically provide weaker consistency guarantees, which may result in some inconsistencies or conflicts between different nodes.	RDBMS provide strong consistency guarantees, which means that all reads and writes are synchronized and the database is always in a consistent state

Some of the features for all the NoSQL data stores are

1. **Consistency**
2. **Transactions**
3. **Query features**
4. **Structure of the data.**

1. **Consistency:**

- Consistency is applicable only for operations on a single key, since these operations are either a get, put, or delete on a single key.
- Optimistic writes can be performed, but are very expensive to implement, because a change in value cannot be determined by the data store.
- In Riak has two ways of resolving update conflicts: either the newest write wins and older writes loose, or both (all) values are returned allowing the client to resolve the conflict.

2. **Transactions:**

- Key-value databases typically do not support traditional transactional semantics such as ACID guarantees.
- However, there are some design patterns that can be used to achieve similar guarantees:
- **Two-phase commit**: This pattern involves using a separate transaction coordinator to manage distributed transactions across multiple key-value stores. The coordinator ensures that all changes are committed or rolled back atomically across all stores.
- **Conditional operations:** This pattern involves using conditional updates to ensure that changes are only made if the current state of the key-value store matches certain criteria.
- **Idempotent operations:** This pattern involves designing operations so that they can be safely retried without causing unintended side effects. For example, if an operation fails due to a network error, it can be safely retried without changing the state of the key-value store.

3. Query Features:

Query features in key-value databases can vary depending on the specific implementation. Here are some common query features you might find in a key-value database:

Key-based lookup: The primary query feature in a key-value database is the ability to retrieve data based on a specific key. This is typically accomplished using a simple "get" operation.

Range queries: Some key-value databases support range queries, which allow you to retrieve a subset of data based on a range of keys. For example, you might retrieve all keys between "A" and "G".

Secondary indexes: Some key-value databases support secondary indexes, which allow you to query data based on attributes other than the primary key. This can be useful for retrieving data based on properties other than the key, such as a user's email address.

MapReduce: Some key-value databases support MapReduce, a programming model for processing large data sets. MapReduce allows you to perform complex data processing tasks across a distributed system.

Full-text search: Some key-value databases support full-text search, which allows you to search for data based on the text within a value. This can be useful for applications that need to search large amounts of text data.

4.Structure of Data: Key-value databases don't care what is stored in the value part of the key-value pair. The value can be a blob, text, JSON, XML, and so on. In Riak, we can use the Content-Type in the POST request to specify the data type.

Suitable Use Cases:

Storing Session Information: Key-value databases can be used to store user session data in web applications, allowing fast and easy access to user session data.

Shopping Cart Data: E-commerce websites have shopping carts tied to the user. As we want the shopping carts to be available all the time, across browsers, machines, and sessions, all the shopping information can be put into the **value** where the key is the **userid**. A Riak cluster would be best suited for these kinds of applications.

User Profiles, Preferences:

- Almost every user has a unique **userId**, username, or some other attribute, as well as preferences such as language, color, timezone, which products the user has access to.
- This can all be put into an object, so getting preferences of a user takes a single **GET** operation. Similarly, product profiles can be stored

Caching: Key-value databases can be used to cache frequently accessed data, reducing the load on the primary data store.

Real-time analytics: Key-value databases can be used to store data for real-time analytics, allowing you to quickly retrieve and analyze data as it is generated.

IoT data storage: Key-value databases can be used to store data from IoT devices, which often generate large volumes of unstructured or semi-structured data that must be accessed quickly.

When Not to Use:

While key-value databases are well-suited for certain use cases, they may not be the best fit for all applications. Here are some scenarios where a key-value database may not be the best choice:

Complex querying: Key-value databases are optimized for fast retrieval of data using a primary key. They are not well-suited for complex querying or joining of data across multiple tables.

Relationships among Data: If you need to have relationships between different sets of data, or correlate the data between different sets of keys, key-value stores are not the best solution to use, even though some key-value stores provide link-walking features.

Multioperation Transactions: If you're saving multiple keys and there is a failure to save any one of them, and you want to revert or roll back the rest of the operations, key-value stores are not the best solution to be used.

ACID compliance: Key-value databases often prioritize performance and scalability over transactional guarantees, which can make them less suitable for applications that require strong consistency and transactional support.

Data validation: Key-value databases do not typically enforce data validation rules or constraints, such as foreign key constraints or unique key constraints.

Data integrity: Key-value databases may not provide the same level of data integrity and durability as relational databases.