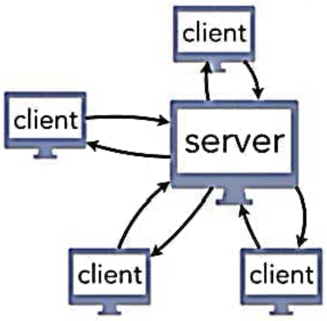
❖ Distribution Models

- ➢ NoSQL's primary driver of interest has been its ability to run databases on a large cluster. As data volumes increase, it becomes more difficult and expensive to scale up—buy a bigger server to run the database on. A more appealing option is to scale out—run the database on a cluster of servers. Aggregate orientation fits well with scaling out because the aggregate is a natural unit to use for distribution.
- ➢ There are two paths to data distribution: replication and sharding. Replication takes the same data and copies it over multiple nodes. Sharding puts different data on different nodes.
- ➢ There are two paths to data distribution: replication and sharding. Replication takes the same data and copies it over multiple nodes. Sharding puts different data on different nodes.

❖ Single Server

- ➢ This model doesn't use any distribution; the database is on a single machine - it handles all the reads and writes. It is easy for operations people to manage and application developers to reason about.
- ➢ Graph databases are the obvious category here—these work best in a single-server configuration.



❖ Shading.

- ➢ A busy data store is busy because different people access different dataset parts. In these circumstances, we can support horizontal scalability by putting different parts of the data onto different servers—a technique that's called **sharding**

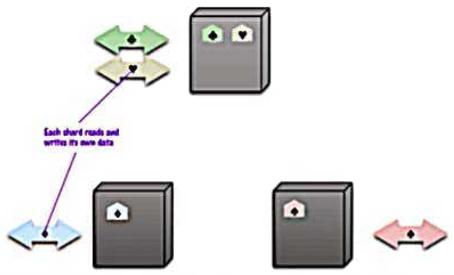


Figure 4.1. Sharding puts different data on separate nodes, each of which does its own reads and writes.

- ➢ In the ideal case, we have different users all talking to different server nodes. Each user only has to talk to one server, so gets rapid responses from that server.
- ➢ Of course the ideal case is a pretty rare beast. In order to get close to it we have to ensure that data that's accessed together is clumped together on the same node and that these clumps are arranged on the nodes to provide the best data access
- ➢ When it comes to arranging the data on the nodes, there are several factors that can help improve performance. If you know that most accesses of certain aggregates are based on a physical location, you can place the data close to where it's being accessed. If you have orders for someone who lives in Boston, you can place that data in your eastern US data center.
- ➢ Many NoSQLdatabases offer auto-sharding, where the database takes on the responsibility of allocating data to shards and ensuring that data access goes to the right shard. This can make it much easier to use sharding in an application.
- ➢ Sharding is particularly valuable for performance because it can improve both read and write performance. Using replication, particularly with caching, can greatly improve read performance but does little for applications that have a lot of writes. Sharding provides a way to horizontally scale writes.
- ➢ Although the data is on different nodes, a node failure makes that shard's data unavailable just as surely as it does for a single-server solution. The resilience benefit it does provide is that only the users of the data on that shard will suffer; however, it's not good to have a database with part of its data missing. With a single server it's easier to pay the effort and cost to keep that server up and running; clusters usually try to use less reliable machines, and you're more likely to get a node failure. So in practice, sharding alone is likely to decrease resilience.

❖ Master Slave Replication

➢ With master-slave distribution, you replicate data across multiple nodes. One node is designated as the master, or primary. This master is the authoritative source for the data and is usually responsible for processing any updates to that data. The other nodes are slaves, or secondaries. A replication process synchronizes the slaves with the master.

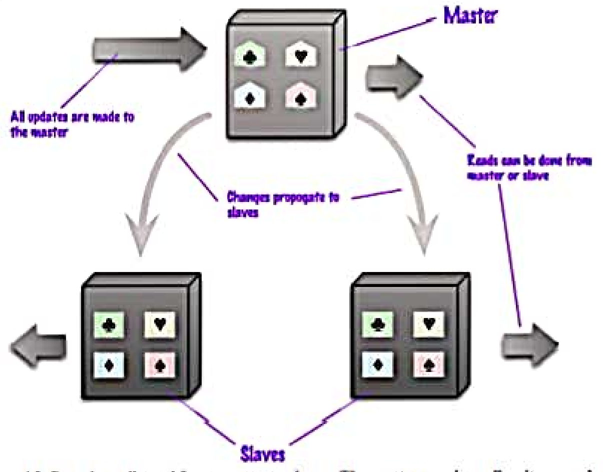

Master
All updates are made to the master
Reads can be done from master or slave
Changes propagate to slaves
Slaves

Figure 4.2. Data is replicated from master to slaves. The master services all writes; reads may come from either master or slaves.

➢ Master-slave replication is most helpful for scaling when you have a read-intensive dataset.

➢ It isn't such a good scheme for datasets with heavy write traffic, although offloading the read traffic will help a bit with handling the write load.

➢ Another advantage of master-slave replication is read resilience: Should the master fail, the slaves can still handle read requests.

➢ The failure of the master does eliminate the ability to handle writes until either the master is restored or a new master is appointed. However, having slaves as replicates of the master does speed up recovery after a failure of the master since a slave can be appointed a new master very quickly.

❖ Peer-to-Peer Replication

➢ 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 attacks these problems by not having a master. All the replicas have equal weight, they can all accept writes, and the loss of any of them doesn't prevent access to the data store.

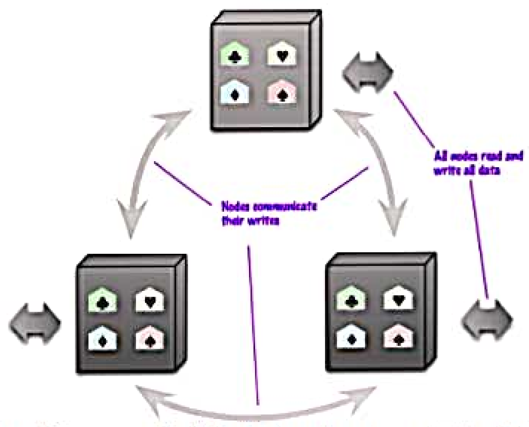

All nodes read and write all data
Nodes communicate their writes

Figure 4.3. Peer-to-peer replication has all nodes applying reads and writes to all the data.

➢ With a peer-to-peer replication cluster, you can ride over node failures without losing access to data. Furthermore, you can easily add nodes to improve your performance.

➢ The biggest complication is, again, consistency. When you can write to two different places, you run the risk that two people will attempt to update the same record at the same time—a write-write conflict. Inconsistencies on read lead to problems but at least they are relatively transient. Inconsistent writes are forever.

2a ❖ **Key-Value Store Features**

➢ Some of the features we will discuss for all the NoSQL data stores are consistency, transactions, query features.

❖ **Consistency**

➢ Consistency is a feature only applicable for operations on a single key in a key-value store. There are various implementations in the key-value store for example in RIAK, the eventually consistent model of consistency is implemented.

➢ In distributed key-value store implementations like Riak, the eventually consistent model of consistency is implemented. Since the value may have already been replicated to other nodes, 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.

➢ Sample code to create a bucket in Riak

```
 Bucket bucket = connection
.createBucket(bucketName)
.withRetrier(attempts(3))
.allowSiblings(siblingsAllowed)
.nVal(numberOfReplicasOfTheData)
.w(numberOfNodesToRespondToWrite)
.r(numberOfNodesToRespondToRead)
.execute();
```

➢ If we need data in every node to be consistent, we can increase the numberOfNodesToRespondToWrite set by w to be the same as nVal. Of course doing that will decrease the write performance of the cluster.

❖ **Transactions:**

➢ In it, there are no guarantees on the writes as many data stores implement transactions in different ways for example RIAK uses the concept of quorum implemented by using the W value replication factor.

➢ Assume we have a Riak cluster with a replication factor of 5 and we supply the W value of 3. When writing, the write is reported as successful only when it is written and reported as a success on at least three of the nodes. This allows Riak to have write tolerance; in our example, with N equal to 5 and with a W value of 3, the cluster can tolerate N - W = 2 nodes being down for write operations, though we would still have lost some data on those nodes for read.

❖ **Query:**

➢ All the key-value stores can be query by the key and that's about it. If we have requirements to query by using some of the attributes of the column, it is not possible for using the database in this condition, our application needs to read the value to recognize if the attribute meets the conditions.

➢ Some key-value databases get around this by providing the ability to search inside the value, such as Riak Search that allows you to query the data just like you would query it using Lucene indexes.

2b ❖ **Suitable Usecases**

➢ Storing Session Information
  - Generally, every web session is unique and is assigned a unique sessionid value. Applications that store the sessionid on disk or in an RDBMS will greatly benefit from moving to a key-value store, since everything about the session can be stored by a single PUT request or retrieved using GET. This single-request operation makes it very fast, as everything about the session is stored in a single object. Solutions such as Memcached are used by many web applications, and Riak can be used when availability is important.

➢ 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, and so on. 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

➢ 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.

❖ **When not to use**

➢ 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.

➢ Query by Data
  - If you need to search the keys based on something found in the value part of the key-value pairs, then key-value stores are not going to perform well for you. There is no way to inspect the value on the database side, with the exception of some products like Riak Search or indexing engines like Lucene [Lucene] or Solr [Solr].

➢ Operations by Sets
  - Since operations are limited to one key at a time, there is no way to operate upon multiple keys at the same time. If you need to operate upon multiple keys, you have to handle this from the client side.

3 Replication in NoSQL databases refers to the process of creating and maintaining multiple copies of the same data on different nodes or servers within a distributed environment. The primary goal of replication is to enhance fault tolerance, availability, and sometimes read scalability. Here are key aspects of replication in NoSQL databases:

❖ **Master Slave Replication**

➢ With master-slave distribution, you replicate data across multiple nodes. One node is designated as the master, or primary. This master is the authoritative source for the data and is usually responsible for processing any updates to that data. The other nodes are slaves, or secondaries. A replication process synchronizes the slaves with the master.

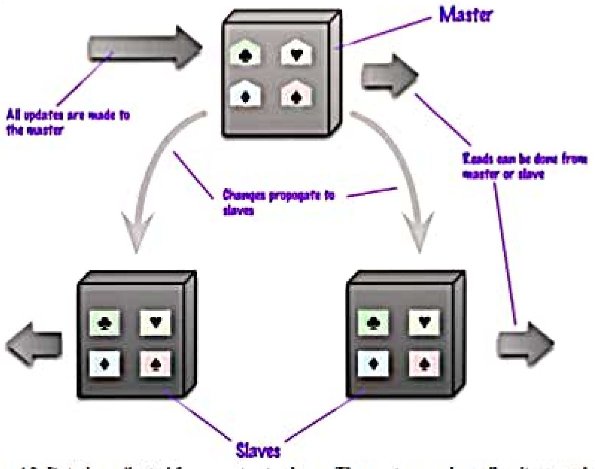

Master
All updates are made to the master
Changes propagate to slaves
Reads can be done from master or slave
Slaves

Figure 4.2. Data is replicated from master to slaves. The master services all writes; reads may come from either master or slaves.

➢ Master-slave replication is most helpful for scaling when you have a read-intensive dataset.

➢ It isn't such a good scheme for datasets with heavy write traffic, although offloading the read traffic will help a bit with handling the write load.

➢ Another advantage of master-slave replication is read resilience: Should the master fail, the slaves can still handle read requests.

➢ The failure of the master does eliminate the ability to handle writes until either the master is restored or a new master is appointed. However, having slaves as replicates of the master does speed up recovery after a failure of the master since a slave can be appointed a new master very quickly.

❖ **Peer-to-Peer Replication**

➢ 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 attacks these problems by not having a master. All the replicas have equal weight, they can all accept writes, and the loss of any of them doesn't prevent access to the data store.

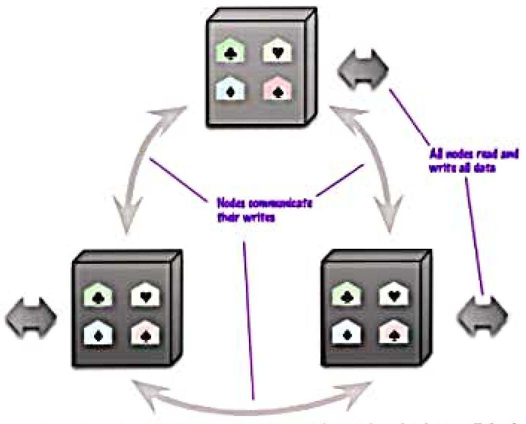

All nodes read and write all data
Nodes communicate their writes

Figure 4.3. Peer-to-peer replication has all nodes applying reads and writes to all the data.

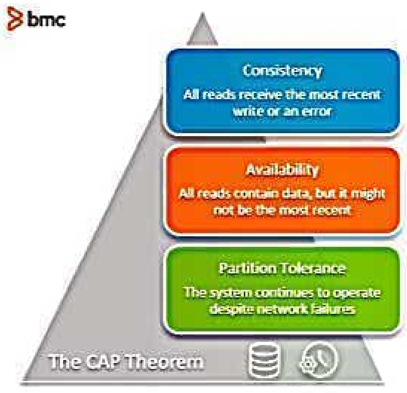
➢ With a peer-to-peer replication cluster, you can ride over node failures without losing access to data. Furthermore, you can easily add nodes to improve your performance.

➢ The biggest complication is, again, consistency. When you can write to two different places, you run the risk that two people will attempt to update the same record at the same time—a write-write conflict. Inconsistencies on read lead to problems but at least they are relatively transient. Inconsistent writes are forever.

❖ CAP Theorem

5b, 6b

- ➢ The CAP theorem maintains that a distributed system can deliver only two of three desired characteristics: consistency, availability, and partition tolerance.
- ➢ Consistency
  - ■ Consistency means that all clients see the same data at the same time, no matter which node they connect to. For this to happen, whenever data is written to one node, it must be instantly forwarded or replicated to all the other nodes in the system before the write is deemed 'successful.'
- ➢ Availability
  - ■ Availability means that any client making a request for data gets a response, even if one or more nodes are down. Another way to state this—all working nodes in the distributed system return a valid response for any request, without exception.
- ➢ Partition tolerance
  - ■ A partition is a communications break within a distributed system—a lost or temporarily delayed connection between two nodes. Partition tolerance means that the cluster must continue to work despite any number of communication breakdowns between nodes in the system.


bmc
Consistency
All reads receive the most recent write or an error
Availability
All reads contain data, but it might not be the most recent
Partition Tolerance
The system continues to operate despite network failures
The CAP Theorem

Key-Value Databases

Now, according to the CAP theorem, a distributed system can achieve at most two out of the three guarantees. Here are three possible scenarios:

- **CA (Consistency and Availability):**
  - In scenarios where partition tolerance is not a concern (no network partitions), a system can prioritize both consistency and availability. This means that all nodes will have a consistent view of the data, and the system will remain available even in the absence of network partitions.
- **CP (Consistency and Partition Tolerance):**
  - In situations where partition tolerance is a priority, a system can prioritize consistency and partition tolerance. This implies that, in the event of a network partition, the system will sacrifice availability to maintain a consistent view of the data across nodes.
- **AP (Availability and Partition Tolerance):**
  - When availability and the ability to tolerate network partitions are crucial, a system can prioritize availability and partition tolerance. In this case, the system may sacrifice consistency, and different nodes might have different views of the data.

❖ What Is a Key-Value Store

6a

- ➢ A key-value store, or key-value database is a simple database that uses an associative array (think of a map or dictionary) as the fundamental data model where each key is associated with one and only one value in a collection. This relationship is referred to as a key-value pair.
- ➢ In each key-value pair the key is represented by an arbitrary string such as a filename, URI or hash. The value can be any kind of data like an image, user preference file or document. The value is stored as a blob requiring no upfront data modeling or schema definition.
- ➢ The storage of the value as a blob removes the need to index the data to improve performance. However, you cannot filter or control what's returned from a request based on the value because the value is opaque.
- ➢ In general, key-value stores have no query language. They provide a way to store, retrieve and update data using simple get, put and delete commands; the path to retrieve data is a direct request to the object in memory or on disk. The simplicity of this model makes a key-value store fast, easy to use, scalable, portable and flexible.

Phone directory

Key	Value
Paul	(091) 9786453778
Greg	(091) 9686154559
Marco	(091) 9868564334

MAC table

Key	Value
10.94.214.172	3c:22:fb:86:c1:b1
10.94.214.173	00:0a:95:9d:68:16
10.94.214.174	3c:1b:fb:45:c4:b1

- ➢ Some of the popular key-value databases are Riak [Riak], Redis (often referred to as Data Structure server) [Redis], Memcached DB and its flavors [Memcached], Berkeley DB [Berkeley DB], HamsterDB (especially suited for embedded use) [HamsterDB], Amazon DynamoDB [Amazon's Dynamo] (not open-source), and Project Voldemort [Project Voldemort] (an open-source implementation of Amazon DynamoDB).
- ➢ For example, all the user information can be stored as an object in the bucket but it may create conflicts; the solution is to break down the object (bucket) into smaller buckets.

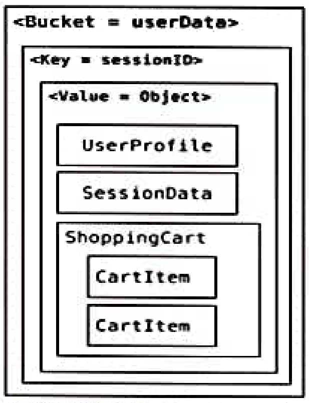


Figure 8.1. Storing all the data in a single bucket

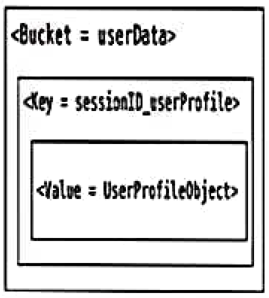


Figure 8.2. Change the key design to segment the data in a single bucket.

❖ When not to use

- ➢ 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.
- ➢ Query by Data
  - ▪ If you need to search the keys based on something found in the value part of the key-value pairs, then key-value stores are not going to perform well for you. There is no way to inspect the value on the database side, with the exception of some products like Riak Search or indexing engines like Lucene [Lucene] or Solr [Solr].
- ➢ Operations by Sets
  - ▪ Since operations are limited to one key at a time, there is no way to operate upon multipl keys at the same time. If you need to operate upon multiple keys, you have to handle this fror the client side.

❖ **Combining Sharding & Replication.**

7b ➢ We can combine 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.

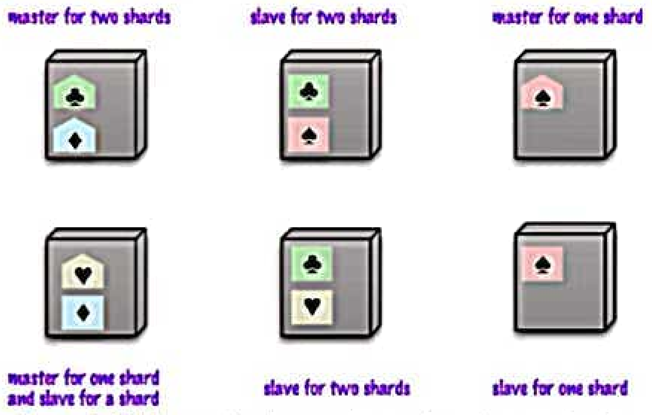


Figure 4.4. Using master-slave replication together with sharding

---

➢ 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

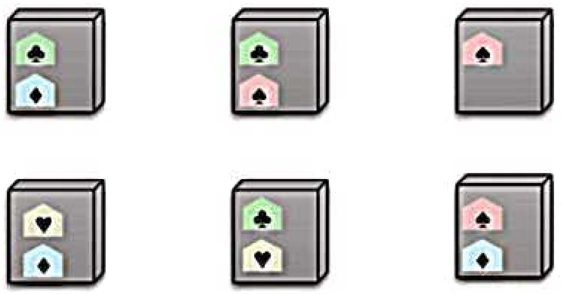


Figure 4.5. Using peer-to-peer replication together with sharding

8a Key-Value Databases and Relational Databases are two different types of database management systems, each with its own set of characteristics, data models, and use cases. Here's a detailed comparison between Key-Value Databases and Relational Databases:

Key-Value Database:

1. **Data Model:**
   - **Structure:** Stores data in a simple key-value pair format. Each data item (value) is associated with a unique identifier (key).
   - **Flexibility:** Offers schema-less or schema-flexible design, allowing for the storage of diverse and dynamic data structures.
2. **Query Language:**
   - **Access:** Typically supports basic CRUD operations (Create, Read, Update, Delete) for key-value pairs.
   - **Limited Query Capabilities:** Querying is often based on key lookups, and complex query operations, especially those involving joins, may be limited.
3. **Use Cases:**
   - **Caching:** Often used for caching frequently accessed data due to its fast key-based retrieval.
   - **Session Storage:** Suitable for storing session-related data in web applications.
   - **Simple Data Storage:** Ideal for scenarios where a straightforward data structure is sufficient.
4. **Scalability:**
   - **Horizontal Scaling:** Designed for horizontal scaling, allowing for the distribution of data across multiple nodes for improved performance and capacity.
5. **Consistency:**
   - **Eventual Consistency:** Many key-value databases prioritize eventual consistency over immediate consistency, especially in distributed environments.
6. **Examples:**
   - Redis, DynamoDB, Riak, Berkeley DB.

Relational Database:

1. **Data Model:**
   - **Structure:** Organizes data into tables with rows and columns, following a predefined schema. Each table has a primary key that uniquely identifies each record.
   - **Enforces Relationships:** Supports relationships between tables through foreign keys, ensuring data integrity.
2. **Query Language:**
   - **Structured Query Language (SQL):** Utilizes SQL for complex query operations, including joins, aggregations, and data manipulation.
   - **Expressive Query Capabilities:** Supports a rich set of operations for retrieving and manipulating data.
3. **Use Cases:**
   - **Transaction Support:** Well-suited for applications requiring ACID properties (Atomicity, Consistency, Isolation, Durability), such as financial transactions.
   - **Complex Queries:** Ideal for scenarios where complex queries, reporting, and analysis are essential.
   - **Structured Data Storage:** Suitable for applications with well-defined and structured data requirements.
4. **Scalability:**
   - **Vertical Scaling:** Traditional relational databases are often scaled vertically by adding more powerful hardware to a single server. Some relational databases also support sharding for horizontal scaling.
5. **Consistency:**
   - **Immediate Consistency:** Emphasizes immediate consistency, ensuring that all transactions follow the ACID properties.
6. **Examples:**
   - MySQL, PostgreSQL, Oracle Database, Microsoft SQL Server.

Comparison Summary:

- **Data Model:** Key-Value databases offer flexibility with a simple key-value pair model, while relational databases provide a structured, table-based model with predefined schemas.
- **Query Language:** Key-Value databases have limited query capabilities, mainly focusing on key lookups, whereas relational databases use SQL for expressive and complex queries.
- **Use Cases:** Key-Value databases are suitable for simple data storage, caching, and scenarios where flexibility is essential. Relational databases are well-suited for applications requiring complex queries, transactions, and structured data.
- **Scalability:** Key-Value databases are designed for horizontal scaling, while traditional relational databases often rely on vertical scaling, though some support sharding for horizontal scaling.
- **Consistency:** Key-Value databases may prioritize eventual consistency, while relational databases emphasize immediate consistency and adhere to ACID properties.

1. **Insert (Put):**

8b
   - **Operation:** Adds a new key-value pair to the database.
   - **Example:**

```python
Database.put("user123", {"name": "John Doe", "age": 30, "email": "j
```

2. **Get (Retrieve):**
   - **Operation:** Retrieves the value associated with a specific key.
   - **Example:**

```python
user_data = Database.get("user123")
```

3. **Update (Put or Update):**
   - **Operation:** Modifies the value associated with an existing key or inserts a new key-value pair if the key does not exist.
   - **Example:**

```python
Database.put("user123", {"name": "John Doe", "age": 31, "email": "j
```

4. **Delete:**
   - **Operation:** Removes a key-value pair from the database.
   - **Example:**

```python
Database.delete("user123")
```

5. **Batch Operations:**
   - **Operation:** Enables the execution of multiple operations in a single batch, improving efficiency.
   - **Example:**

```python
batch_operations = [
    {"operation": "put", "key": "item1", "value": "value1"},
    {"operation": "put", "key": "item2", "value": "value2"},
    {"operation": "delete", "key": "item3"}
]
Database.batch(batch_operations)
```

6. **Increment/Decrement:**
   - **Operation:** Atomically increments or decrements the value associated with a numeric key.
   - **Example:**

```python
Database.increment("counter", 1)  # Increment the counter by 1
```

7. **Existence Check:**
   - **Operation:** Checks whether a key exists in the database.
   - **Example:**

```python
exists = Database.exists("user123")
```

8. **Range Queries (Scan):**
   - **Operation:** Retrieves a range of key-value pairs within a specified range or pattern.
   - **Example:**

```python
results = Database.scan("user", "user999")  # Get all keys starting
```

9. **TTL (Time-to-Live):**
   - **Operation:** Sets a time-to-live for a key-value pair, specifying the duration after which the pair will be automatically deleted.
   - **Example:**

```python
Database.put_with_ttl("session123", {"user_id": "user123"}, ttl_seco
```

10. **Conditional Put:**
    - **Operation:** Updates the value associated with a key only if certain conditions are met (e.g., if the current value matches a specified condition).
    - **Example:**

```python
Database.put_if_condition("user123", {"name": "Jane Doe"}, conditio
```