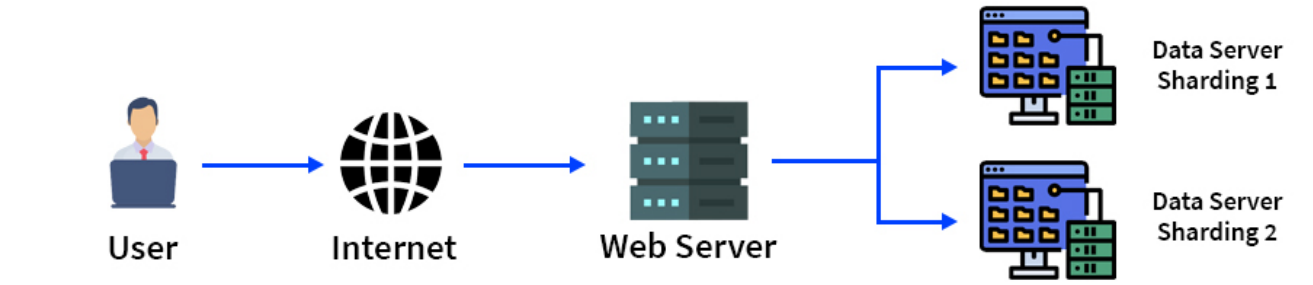
## What is Sharding or Data Partitioning?

Take the example of ***Pizza*** (*yes!!! your favorite food*). You get the pizza in different slices and you share these slices with your friends. **Sharding** which is also known as **data partitioning** works on the same concept of sharing the Pizza slices. It is basically a database architecture pattern in which we split a large dataset into smaller chunks (logical shards) and we store/distribute these chunks in different machines/database nodes (physical shards). Each chunk/partition is known as a "**shard**" and each shard has the same database schema as the original database. We distribute the data in such a way that each row appears in exactly one shard. It's a good mechanism to improve the **scalability** of an application.


No Sharding




```
NOTE:Database shards are autonomous; they don't share any of the same data or computing
resources. In some cases, though, it may make sense to replicate certain tables into each
shard to serve as reference tables.
```

---

## Advantages of Sharding

- **Solve Scalability Issue:** With a single database server architecture any application experience performance degradation when users start growing on that application.  Reads and write queries become slower and the network bandwidth starts to saturate.  At some point, you will be running out of disk space. Database sharding fixes all these issues by partitioning the data across multiple machines.
- **High Availability:** A problem with single server architecture is that if an outage happens then the entire application will be unavailable which is not good for a website with more number of users. This is not the case with a sharded database. If an outage happens in sharded architecture, then only some specific shards will be down. All the other shards will continue the operation and the entire application won't be unavailable for the users.

- **Speed Up Query Response Time:** When you submit a query in an application with a large monolithic database and have no sharded architecture, it takes more time to find the result. It has to search every row in the table and that slows down the response time for the query you have given. This doesn't happen in sharded architecture. In a sharded database a query has to go through fewer rows and you receive the response in less time.
- **More Write Bandwidth:** For many applications writing is a major bottleneck. With no master database serializing writes sharded architecture allows you to write in parallel and increase your write throughput.
- **Scaling Out:** Sharding a database facilitates ***horizontal scaling***, known as *scaling out*. In horizontal scaling, you **add more machines** in the network and distribute the load on these machines for faster processing and response. This has many advantages. You can do more work simultaneously and you can handle high requests from the users, especially when writing data because there are parallel paths through your system. You can also load balance web servers that access shards over different network paths, which are processed by different CPUs, and use separate caches of RAM or disk IO paths to process work.

## Disadvantages of Sharding

- **Adds Complexity in the System:** You need to be careful while implementing a proper sharded database architecture in an application. It's a complicated task and if it's not implemented properly then you may lose the data or get corrupted tables in your database. You also need to manage the data from multiple shard locations instead of managing and accessing it from a single entry point. This may affect the workflow of your team which can be potentially disruptive to some teams.
- **Rebalancing Data:** In a sharded database architecture, sometimes shards become unbalanced (when a shard outgrows other shards). Consider an example that you have two shards of a database. One shard store the name of the customers begins with letter A through M. Another shard store the name of the customer begins with the letters N through Z. If there are so many users with the letter L then shard one will have more data than shard two. This will affect the performance (slow down) of the application and it will stall out for a significant portion of your users. The A-M shard will become unbalance and it will be known as ***database hotspot***. To overcome this problem and to rebalance the data you need to do re-sharding for even data distribution. Moving data from one shard to another shard is not a good idea because it requires a lot of downtimes.
- **Joining Data From Multiple Shards is Expensive:** In a single database, joins can be performed easily to implement any functionalities. But in sharded architecture, you need to pull the data from different shards and you need to perform joins across multiple networked servers You can't submit a single query to get the data from various shards. You need to submit **multiple queries** for each one of the shards, pull out the data, and join the data across the network. This is going to be a very expensive and time-consuming process. It adds **latency** to your system.
- **No Native Support:** Sharding is not natively supported by every database engine. For example, PostgreSQL doesn't include automatic sharding features, so there you have to do manual sharding. You need to follow the "roll-your-own" approach. It will be difficult for you to find the tips or documentation for sharding and troubleshoot the problem during the implementation of sharding.

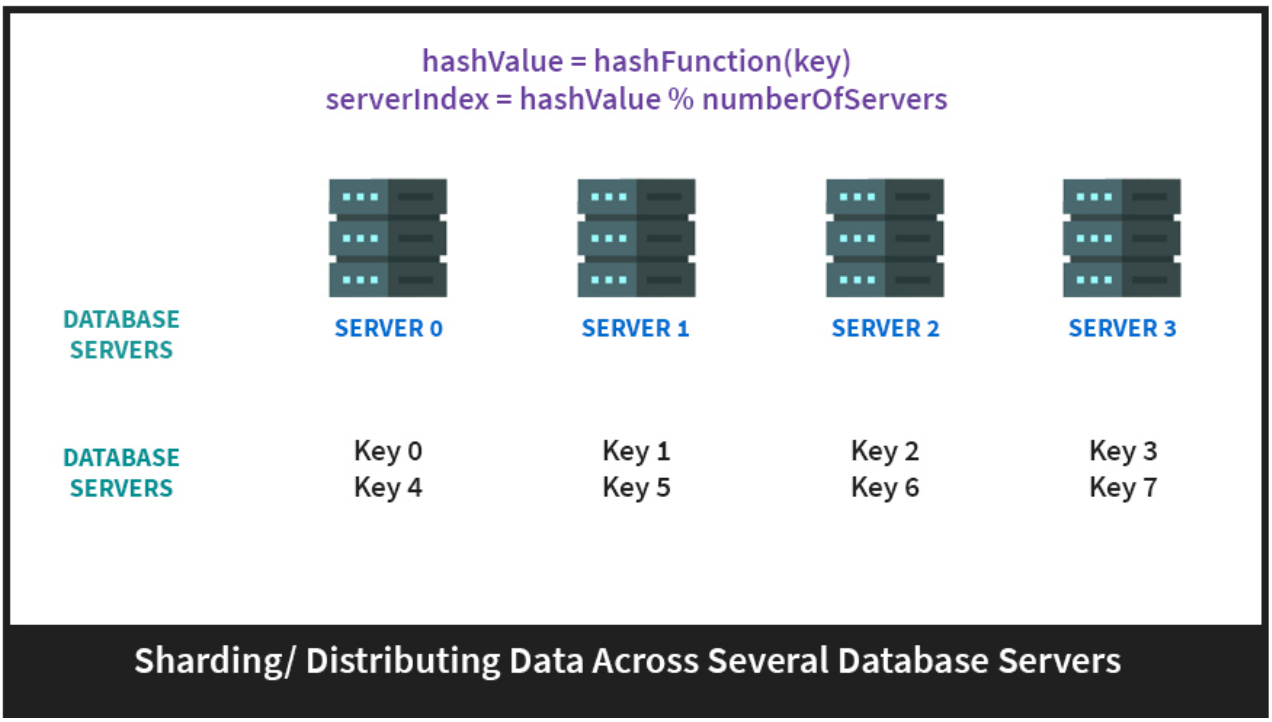
---

## Types of sharding architectures

### 1. Key Based Sharding

This technique is also known as **hash-based** sharding. Here, we take the value of an entity such as customer ID, customer email, IP address of a client, zip code, etc and we use this value as an input of the **hash function**. This process generates a **hash value** which is used to determine which shard we need to use to store the data. We need to keep in mind that the values entered into the hash function should all come from the **same column** (shard key) just to ensure that data is placed in the correct order and in a consistent manner. Basically, shard keys act like a *primary key* or a unique identifier for individual rows.

Consider an example that you have 3 database servers and each request has an application id which is incremented by 1 every time a new application is registered. To determine which server data should be placed on, we perform a modulo operation on these applications id with the number 3. Then the remainder is used to identify the server to store our data.



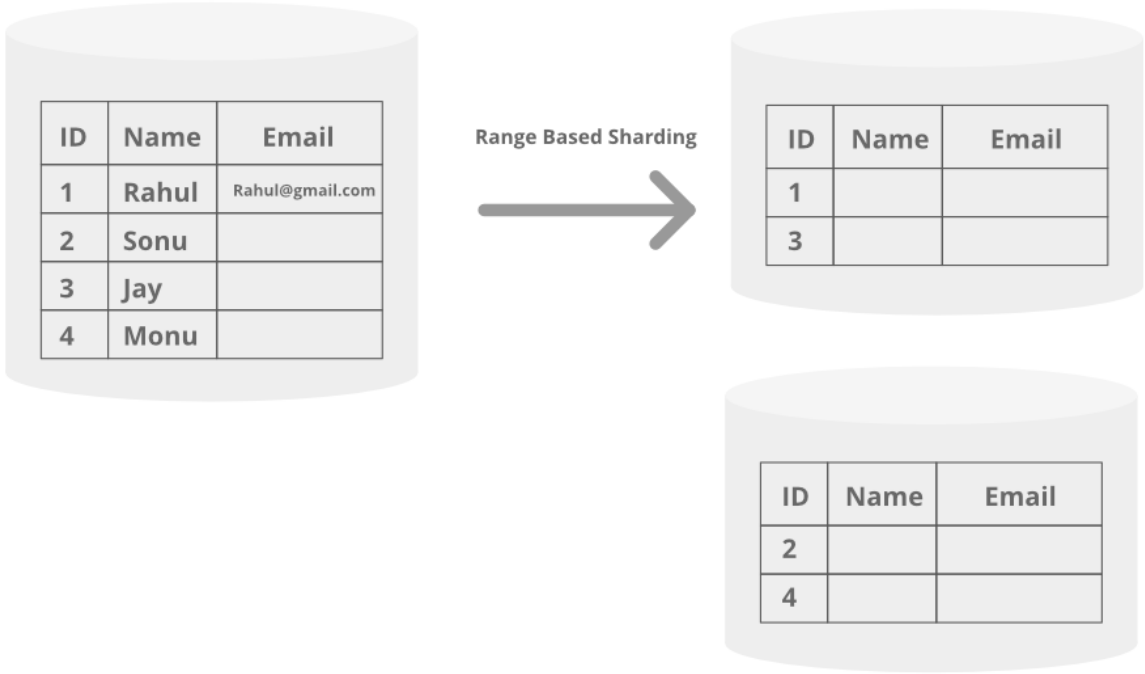

The downside of this method is **elastic load balancing** which means if you will try to add or remove the database servers dynamically it will be a difficult and expensive process. For example, in the above one if you will add 5 more servers then you need to add more corresponding hash values for the additional entries. Also, the majority of the existing keys need to be remapped to their new, correct hash value and then migrated to a new server. The hash function needs to be changed from modulo 3 to modulo 8. While the migration of data is in effect both the new and old hash functions won't be valid. During the migration, your application won't be able to service a large number of requests and you'll experience downtime for your application till the migration completes.

**Note:** *A shard shouldn't contain values that might change over time. It should be always static otherwise it will slow down the performance.*

---

## 2. Horizontal or Range Based Sharding

In this method, we split the data based on the **ranges** of a given value inherent in each entity. Let's say you have a database of your online customers' names and email information. You can split this information into two shards. In one shard you can keep the info of customers whose first name starts with A-P and in another shard, keep the information of the rest of the customers.



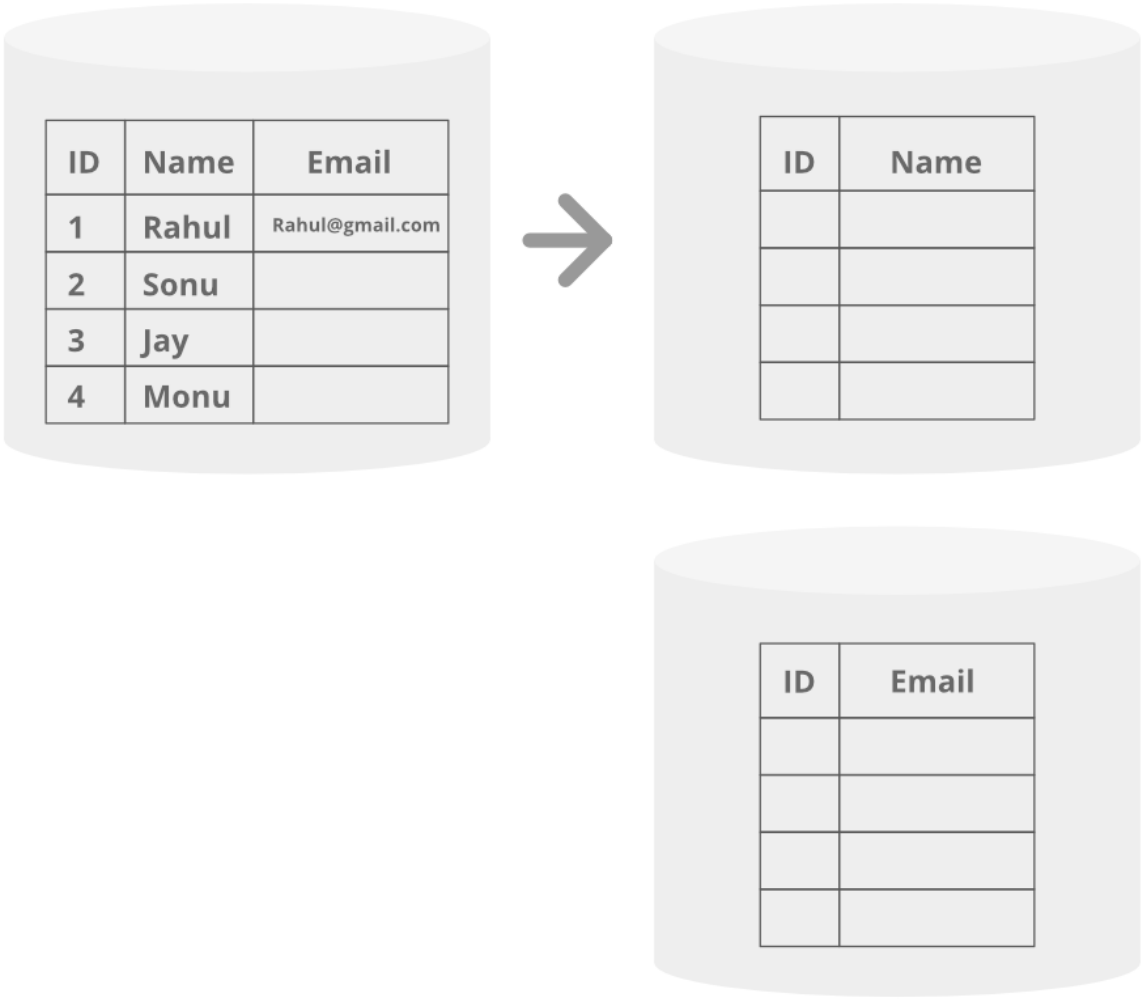

Range-based sharding is the simplest sharding method to implement. Every shard holds a different set of data but they all have the same schema as the original database. In this method, you just need to identify in which range your data falls, and then you can store the entry to the corresponding shard. This method is best suitable for storing non-static data (example: storing the contact info for students in a college.)

The drawback of this method is that the data may not be **evenly distributed** on shards. In the above example, you might have a lot of customers whose names fall into the category of A-P. In such cases, the first shard will have to take more load than the second one and it can become a system bottleneck.

---

## 3. Vertical Sharding

In this method, we split the entire column from the table and we put those columns into new distinct tables. Data is totally independent of one partition to the other ones. Also, each partition holds both distinct rows and columns. Take the example of Twitter features. We can split different features of an entity in different shards on different machines. On Twitter users might have a profile, number of followers, and some tweets posted by his/her own. We can place the user profiles on one shard, followers in the second shard, and tweets on a third shard.



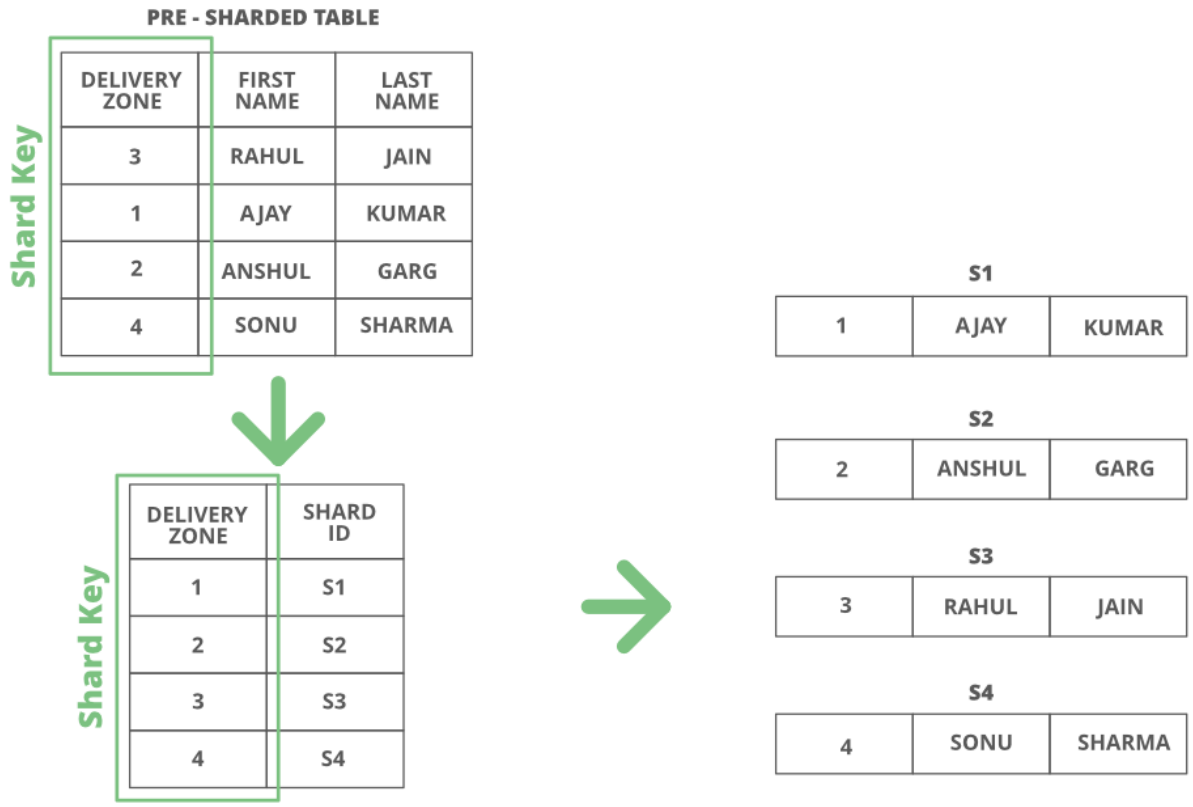

In this method, you can separate and handle the critical part (for example user profiles) non-critical part of your data (for example, blog posts) individually and build different replication and consistency models around it. This is one of the main advantages of this method.

The main drawback of this scheme is that to answer some queries you may have to combine the data from different shards which unnecessarily increases the development and operational complexity of the system. Also, if your application will grow later and you add some more features in it then you will have to further shard a feature-specific database across multiple servers.

---

## 4. Directory-Based Sharding(application -> lookuptable (DS sharding)-> directed to DB server)

In this method, we create and maintain a **lookup service** or lookup table for the original database. Basically we use a **shard key** for lookup table and we do **mapping** for each entity that exists in the database. This way we keep track of which database shards hold which data.

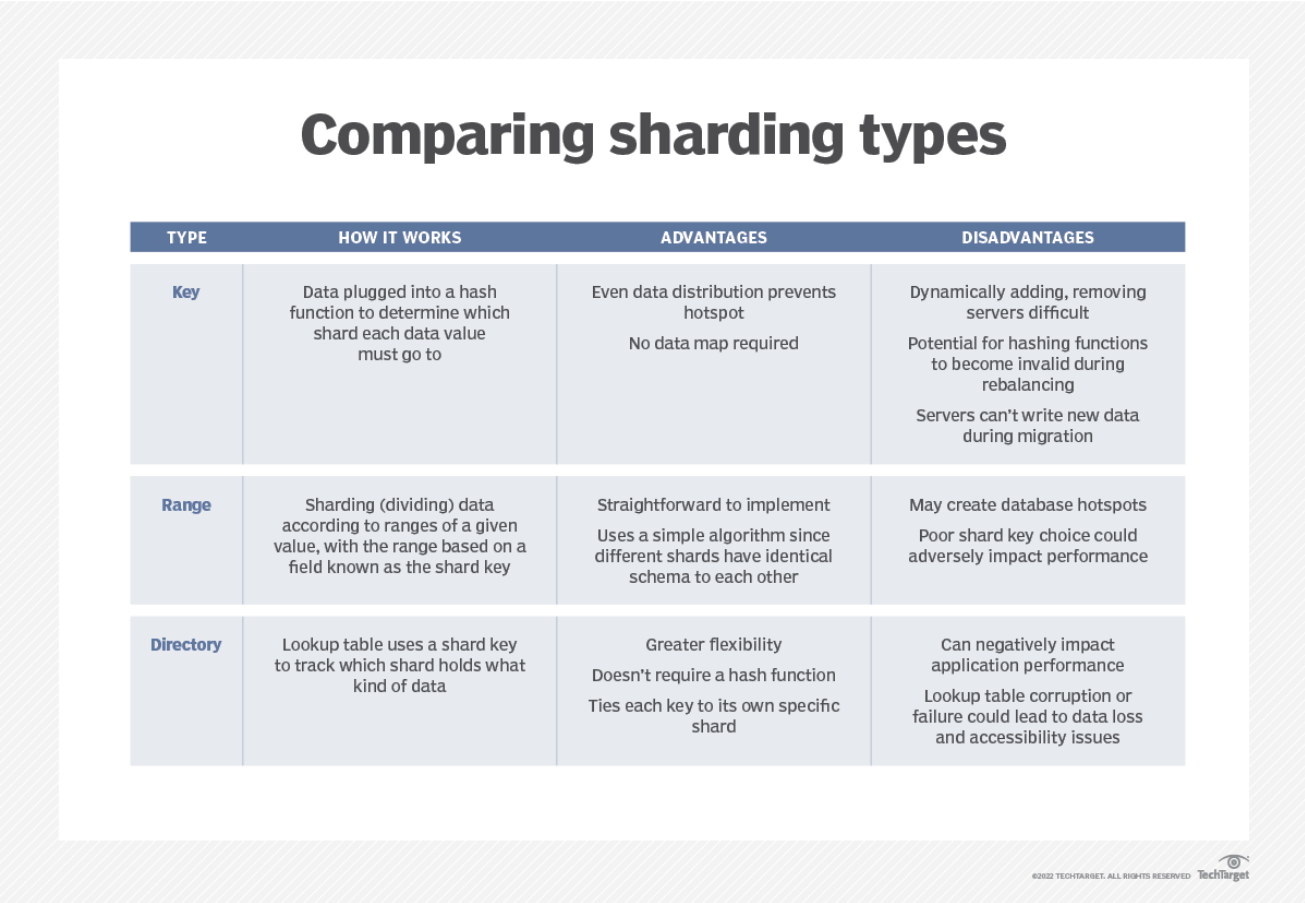



The lookup table holds a static set of information about where specific data can be found. In the above image, you can see that we have used the delivery zone as a shard key. Firstly the client application queries the lookup service to find out the shard (database partition) on which the data is placed. When the lookup service returns the shard it queries/updates that shard.

Directory-based sharding is much more flexible than range based and key-based sharding. In range-based sharding, you're bound to specify the ranges of values. In key-based, you are bound to use a fixed hash function which is difficult to change later. In this approach, you're free to use any algorithm you want to assign to data entries to shards. Also, it's easy to add shards dynamically in this approach.

The major drawback of this approach is the single point of failure of the lookup table. If it will be corrupted or failed then it will impact writing new data or accessing existing data from the table.

---



## Comparing sharding types

| TYPE | HOW IT WORKS | ADVANTAGES | DISADVANTAGES |
|---|---|---|---|
| Key | Data plugged into a hash function to determine which shard each data value must go to | Even data distribution prevents hotspot<br>No data map required | Dynamically adding, removing servers difficult<br>Potential for hashing functions to become invalid during rebalancing<br>Servers can't write new data during migration |
| Range | Sharding (dividing) data according to ranges of a given value, with the range based on a field known as the shard key | Straightforward to implement<br>Uses a simple algorithm since different shards have identical schema to each other | May create database hotspots<br>Poor shard key choice could adversely impact performance |
| Directory | Lookup table uses a shard key to track which shard holds what kind of data | Greater flexibility<br>Doesn't require a hash function<br>Ties each key to its own specific shard | Can negatively impact application performance<br>Lookup table corruption or failure could lead to data loss and accessibility issues |



---

## Conclusion

Sharding is a great solution when the single database of your application is not capable to handle/store a huge amount of growing data. Sharding helps to scale the database and improve the performance of the application. However, it also adds some complexity to your system.

---