

<https://www.digitalocean.com/community/tutorials/understanding-database-sharding>
<https://www.mongodb.com/features/database-sharding-explained>

Introduction

Any application or website that sees significant growth will eventually need to scale in order to accommodate increases in traffic. For data-driven applications and websites, it's critical that scaling is done in a way that ensures the security and integrity of their data. It can be difficult to predict how popular a website or application will become or how long it will maintain that popularity, which is why some organizations choose a database architecture that allows them to scale their databases dynamically.

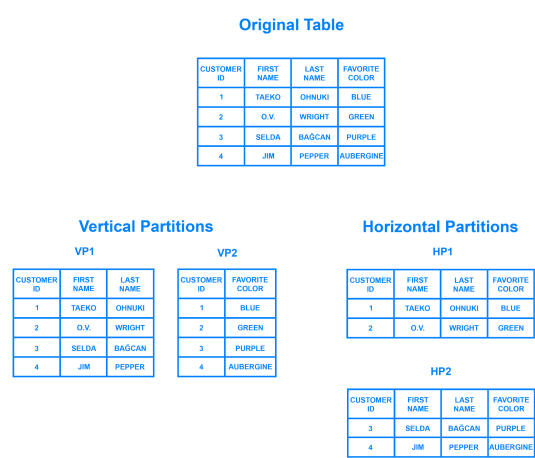
In this conceptual article, we will discuss one such database architecture: *sharded databases*. Sharding has been receiving lots of attention in recent years, but many don't have a clear understanding of what it is or the scenarios in which it might make sense to shard a database. We will go over what sharding is, some of its main benefits and drawbacks, and also a few common sharding methods.

What is Sharding?

Sharding is a database architecture pattern related to *horizontal partitioning* — the practice of separating one table's rows into multiple different tables, known as partitions. Each partition has the same schema and columns, but also entirely different rows. Likewise, the data held in each is unique and independent of the data held in other partitions.

It can be helpful to think of horizontal partitioning in terms of how it relates to *vertical partitioning*. In a vertically-partitioned table, entire columns are separated out and put into

new, distinct tables. The data held within one vertical partition is independent from the data in all the others, and each holds both distinct rows and columns. The following diagram illustrates how a table could be partitioned both horizontally and vertically:



Sharding involves breaking up one’s data into two or more smaller chunks, called *logical shards*. The logical shards are then distributed across separate database nodes, referred to as *physical shards*, which can hold multiple logical shards. Despite this, the data held within all the shards collectively represent an entire logical dataset.

Database shards exemplify a *shared-nothing architecture*. This means that the 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. For example, let’s say there’s a database for an application that depends on fixed conversion rates for weight measurements. By replicating a table containing the necessary conversion rate data into each shard, it would help to ensure that all of the data required for queries is held in every shard.

Oftentimes, sharding is implemented at the application level, meaning that the application includes code that defines which shard to transmit reads and writes to. However, some database management systems have sharding capabilities built in, allowing you to implement sharding directly at the database level.

Given this general overview of sharding, let's go over some of the positives and negatives associated with this database architecture.

Benefits of Sharding

The main appeal of sharding a database is that it can help to facilitate *horizontal scaling*, also known as *scaling out*. Horizontal scaling is the practice of adding more machines to an existing stack in order to spread out the load and allow for more traffic and faster processing. This is often contrasted with *vertical scaling*, otherwise known as *scaling up*, which involves upgrading the hardware of an existing server, usually by adding more RAM or CPU.

It's relatively simple to have a relational database running on a single machine and scale it up as necessary by upgrading its computing resources. Ultimately, though, any non-distributed database will be limited in terms of storage and compute power, so having the freedom to scale horizontally makes your setup far more flexible.

Another reason why some might choose a sharded database architecture is to speed up query response times. When you submit a query on a database that hasn't been sharded, it may have to search every row in the table you're querying before it can find the result set you're looking for. For an application with a large, monolithic database, queries can become

prohibitively slow. By sharding one table into multiple, though, queries have to go over fewer rows and their result sets are returned much more quickly.

Sharding can also help to make an application more reliable by mitigating the impact of outages. If your application or website relies on an unsharded database, an outage has the potential to make the entire application unavailable. With a sharded database, though, an outage is likely to affect only a single shard. Even though this might make some parts of the application or website unavailable to some users, the overall impact would still be less than if the entire database crashed.

Drawbacks of Sharding

While sharding a database can make scaling easier and improve performance, it can also impose certain limitations. Here, we'll discuss some of these and why they might be reasons to avoid sharding altogether.

The first difficulty that people encounter with sharding is the sheer complexity of properly implementing a sharded database architecture. If done incorrectly, there's a significant risk that the sharding process can lead to lost data or corrupted tables. Even when done correctly, though, sharding is likely to have a major impact on your team's workflows. Rather than accessing and managing one's data from a single entry point, users must manage data across multiple shard locations, which could potentially be disruptive to some teams.

One problem that users sometimes encounter after having sharded a database is that the shards eventually become unbalanced. By way of example, let's say you have a database with two separate shards, one for customers whose last names begin with letters A through

M and another for those whose names begin with the letters N through Z. However, your application serves an inordinate amount of people whose last names start with the letter G. Accordingly, the A-M shard gradually accrues more data than the N-Z one, causing the application to slow down and stall out for a significant portion of your users. The A-M shard has become what is known as a *database hotspot*. In this case, any benefits of sharding the database are canceled out by the slowdowns and crashes. The database would likely need to be repaired and resharded to allow for a more even data distribution.

Another major drawback is that once a database has been sharded, it can be very difficult to return it to its unsharded architecture. Any backups of the database made before it was sharded won't include data written since the partitioning. Consequently, rebuilding the original unsharded architecture would require merging the new partitioned data with the old backups or, alternatively, transforming the partitioned DB back into a single DB, both of which would be costly and time consuming endeavors.

A final disadvantage to consider is that sharding isn't natively supported by every database engine. For instance, PostgreSQL does not include automatic sharding as a feature, although it is possible to manually shard a PostgreSQL database. There are a number of Postgres forks that do include automatic sharding, but these often trail behind the latest PostgreSQL release and lack certain other features. Some specialized database technologies — like MySQL Cluster or certain database-as-a-service products like MongoDB Atlas — do include auto-sharding as a feature, but vanilla versions of these database management systems do not. Because of this, sharding often requires a “roll your own” approach. This means that documentation for sharding or tips for troubleshooting problems are often difficult to find.

These are, of course, only some general issues to consider before sharding. There may be many more potential drawbacks to sharding a database depending on its use case.

Now that we've covered a few of sharding's drawbacks and benefits, we will go over a few different architectures for sharded databases.

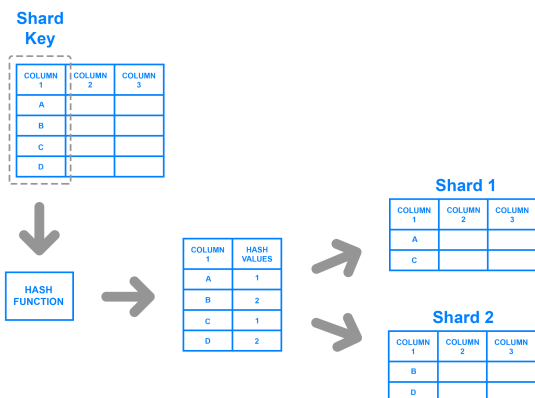
Sharding Architectures

Once you've decided to shard your database, the next thing you need to figure out is how you'll go about doing so. When running queries or distributing incoming data to sharded tables or databases, it's crucial that it goes to the correct shard. Otherwise, it could result in lost data or painfully slow queries. In this section, we'll go over a few common sharding architectures, each of which uses a slightly different process to distribute data across shards.

Key Based Sharding

Key based sharding, also known as *hash based sharding*, involves using a value taken from newly written data — such as a customer's ID number, a client application's IP address, a ZIP code, etc. — and plugging it into a *hash function* to determine which shard the data should go to. A hash function is a function that takes as input a piece of data (for example, a customer email) and outputs a discrete value, known as a *hash value*. In the case of sharding, the hash value is a shard ID used to determine which shard the incoming data will be stored on.

Altogether, the process looks like this:



To ensure that entries are placed in the correct shards and in a consistent manner, the values entered into the hash function should all come from the same column. This column is known as a *shard key*. In simple terms, shard keys are similar to *primary keys* in that both are columns which are used to establish a unique identifier for individual rows. Broadly speaking, a shard key should be static, meaning it shouldn't contain values that might change over time. Otherwise, it would increase the amount of work that goes into update operations, and could slow down performance.

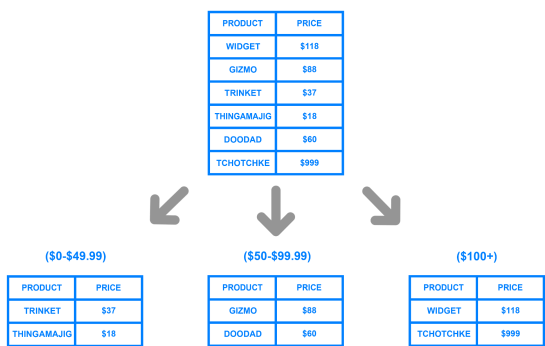
While key based sharding is a fairly common sharding architecture, it can make things tricky when trying to dynamically add or remove additional servers to a database. As you add servers, each one will need a corresponding hash value and many of your existing entries, if not all of them, will need to be remapped to their new, correct hash value and then migrated to the appropriate server. As you begin rebalancing the data, neither the new nor the old hashing functions will be valid. Consequently, your server won't be able to write any new data during the migration and your application could be subject to downtime.

The main appeal of this strategy is that it can be used to evenly distribute data so as to prevent hotspots. Also, because it distributes data algorithmically, there's no need to

maintain a map of where all the data is located, as is necessary with other strategies like range or directory based sharding.

Range Based Sharding

Range based sharding involves sharding data based on ranges of a given value. To illustrate, let's say you have a database that stores information about all the products within a retailer's catalog. You could create a few different shards and divvy up each products' information based on which price range they fall into, like this:



The main benefit of range based sharding is that it's relatively simple to implement. Every shard holds a different set of data but they all have an identical schema as one another, as well as the original database. The application code just reads which range the data falls into and writes it to the corresponding shard.

On the other hand, range based sharding doesn't protect data from being unevenly distributed, leading to the aforementioned database hotspots. Looking at the example diagram, even if each shard holds an equal amount of data the odds are that specific products will receive more attention than others. Their respective shards will, in turn, receive a disproportionate number of reads.

Here, the **Delivery Zone** column is defined as a shard key. Data from the shard key is written to the lookup table along with whatever shard each respective row should be written to. This is similar to range based sharding, but instead of determining which range the shard key's data falls into, each key is tied to its own specific shard. Directory based sharding is a good choice over range based sharding in cases where the shard key has a low cardinality and it doesn't make sense for a shard to store a range of keys. Note that it's also distinct from key based sharding in that it doesn't process the shard key through a hash function; it just checks the key against a lookup table to see where the data needs to be written.

The main appeal of directory based sharding is its flexibility. Range based sharding architectures limit you to specifying ranges of values, while key based ones limit you to using a fixed hash function which, as mentioned previously, can be exceedingly difficult to change later on. Directory based sharding, on the other hand, allows you to use whatever system or algorithm you want to assign data entries to shards, and it's relatively easy to dynamically add shards using this approach.

While directory based sharding is the most flexible of the sharding methods discussed here, the need to connect to the lookup table before every query or write can have a detrimental impact on an application's performance. Furthermore, the lookup table can become a single point of failure: if it becomes corrupted or otherwise fails, it can impact one's ability to write new data or access their existing data.

Should I Shard?

Whether or not one should implement a sharded database architecture is almost always a matter of debate. Some see sharding as an inevitable outcome for databases that reach a

certain size, while others see it as a headache that should be avoided unless it's absolutely necessary, due to the operational complexity that sharding adds.

Because of this added complexity, sharding is usually only performed when dealing with very large amounts of data. Here are some common scenarios where it may be beneficial to shard a database:

- The amount of application data grows to exceed the storage capacity of a single database node.
- The volume of writes or reads to the database surpasses what a single node or its read replicas can handle, resulting in slowed response times or timeouts.
- The network bandwidth required by the application outpaces the bandwidth available to a single database node and any read replicas, resulting in slowed response times or timeouts.

Before sharding, you should exhaust all other options for optimizing your database. Some optimizations you might want to consider include:

- **Setting up a remote database.** If you're working with a monolithic application in which all of its components reside on the same server, you can improve your database's performance by moving it over to its own machine. This doesn't add as much complexity as sharding since the database's tables remain intact. However, it still allows you to vertically scale your database apart from the rest of your infrastructure.
- **Implementing caching.** If your application's read performance is what's causing you trouble, caching is one strategy that can help to improve it. Caching involves temporarily storing data that has already been requested in memory, allowing you to access it much more quickly later on.
- **Creating one or more read replicas.** Another strategy that can help to improve read performance, this involves copying the data from one database server (the *primary server*) over to one or more *secondary servers*. Following this, every new write goes to the primary before being copied over to the secondaries, while reads are made exclusively to the secondary servers. Distributing reads and writes like this keeps any one machine from taking

on too much of the load, helping to prevent slowdowns and crashes. Note that creating read replicas involves more computing resources and thus costs more money, which could be a significant constraint for some.

- **Upgrading to a larger server.** In most cases, scaling up one's database server to a machine with more resources requires less effort than sharding. As with creating read replicas, an upgraded server with more resources will likely cost more money. Accordingly, you should only go through with resizing if it truly ends up being your best option.

Bear in mind that if your application or website grows past a certain point, none of these strategies will be enough to improve performance on their own. In such cases, sharding may indeed be the best option for you.

Conclusion

Sharding can be a great solution for those looking to scale their database horizontally.

However, it also adds a great deal of complexity and creates more potential failure points for your application. Sharding may be necessary for some, but the time and resources needed to create and maintain a sharded architecture could outweigh the benefits for others.

What Is Database Sharding?

Sharding is a method for distributing a single dataset across multiple databases, which can then be stored on multiple machines. This allows for larger datasets to be split in smaller chunks and stored in multiple data nodes, increasing the total storage capacity of the system. See more on the [basics of sharding](#) here.

Similarly, by distributing the data across multiple machines, a sharded database can handle more requests than a single machine can.

Sharding is a form of scaling known as horizontal scaling or scale-out, as additional nodes are brought on to share the load. Horizontal scaling allows for near-limitless scalability to handle big data and intense workloads. In contrast, vertical scaling refers to increasing the power of a single machine or single server through a more powerful CPU, increased RAM, or increased storage capacity.

Do You Need Database Sharding?

Database sharding, as with any distributed architecture, does not come for free. There is overhead and complexity in setting up shards, maintaining the data on each shard, and properly routing requests across those shards. Before you begin sharding, consider if one of the following alternative solutions will work for you.

Vertical Scaling

By simply upgrading your machine, you can scale vertically without the complexity of sharding. Adding RAM, upgrading your computer (CPU), or increasing the storage available to your database are simple solutions that do not require you to change the design of either your database architecture or your application.

Specialized Services or Databases

Depending on your use case, it may make more sense to simply shift a subset of the burden onto other providers or even a separate database. For example, blob or file storage can be moved directly to a cloud provider such as Amazon S3. Analytics or full-text search can be handled by specialized services or a data warehouse. Offloading this particular functionality can make more sense than trying to shard your entire database.

Replication

If your data workload is primarily read-focused, replication increases availability and read performance while avoiding some of the complexity of database sharding. By

simply spinning up additional copies of the database, read performance can be increased either through load balancing or through geo-located query routing. However, replication introduces complexity on write-focused workloads, as each write must be copied to every replicated node.

On the other hand, if your core application database contains large amounts of data, requires high read and high write volume, and/or you have specific availability requirements, a sharded database may be the way forward. Let's look at the advantages and disadvantages of sharding.

Advantages of Sharding

Sharding allows you to scale your database to handle increased load to a nearly unlimited degree by providing increased read/write throughput, storage capacity, and high availability. Let's look at each of those in a little more detail.

- **Increased Read/Write Throughput** — By distributing the dataset across multiple shards, both read and write operation capacity is increased as long as read and write operations are confined to a single shard.
- **Increased Storage Capacity** — Similarly, by increasing the number of shards, you can also increase overall total storage capacity, allowing near-infinite scalability.
- **High Availability** — Finally, shards provide high availability in two ways. First, since each shard is a replica set, every piece of data is replicated. Second, even if an entire shard becomes unavailable since the data is distributed, the database as a whole still remains partially functional, with part of the schema on different shards.

Disadvantages of Sharding

Sharding does come with several drawbacks, namely overhead in query result compilation, complexity of administration, and increased infrastructure costs.

- Query Overhead — Each sharded database must have a separate machine or service which understands how to route a querying operation to the appropriate shard. This introduces additional latency on every operation. Furthermore, if the data required for the query is horizontally partitioned across multiple shards, the router must then query each shard and merge the result together. This can make an otherwise simple operation quite expensive and slow down response times.
- Complexity of Administration — With a single unsharded database, only the database server itself requires upkeep and maintenance. With every sharded database, on top of managing the shards themselves, there are additional service nodes to maintain. Plus, in cases where replication is being used, any data updates must be mirrored across each replicated node. Overall, a sharded database is a more complex system which requires more administration.
- Increased Infrastructure Costs — Sharding by its nature requires additional machines and compute power over a single database server. While this allows your database to grow beyond the limits of a single machine, each additional shard comes with higher costs. The cost of a distributed database system, especially if it is missing the proper optimization, can be significant.

Having considered the pros and cons, let's move forward and discuss implementation.

How Does Sharding Work?

In order to shard a database, we must answer several fundamental questions. The answers will determine your implementation.

First, how will the data be distributed across shards? This is the fundamental question behind any sharded database. The answer to this question will have effects on both performance and maintenance. More detail on this can be found in the “Sharding Architectures and Types” section.

Second, what types of queries will be routed across shards? If the workload is primarily read operations, replicating data will be highly effective at increasing performance, and

you may not need sharding at all. In contrast, a mixed read-write workload or even a primarily write-based workload will require a different architecture.

Finally, how will these shards be maintained? Once you have sharded a database, over time, data will need to be redistributed among the various shards, and new shards may need to be created. Depending on the distribution of data, this can be an expensive process and should be considered ahead of time.

With these questions in mind, let's consider some sharding architectures.

Sharding Architectures and Types

While there are many different sharding methods, we will consider four main kinds: ranged/dynamic sharding, algorithmic/hashed sharding, entity/relationship-based sharding, and geography-based sharding.

Ranged/Dynamic Sharding

Ranged sharding, or dynamic sharding, takes a field on the record as an input and, based on a predefined range, allocates that record to the appropriate shard. Ranged sharding requires there to be a lookup table or service available for all queries or writes. For example, consider a set of data with IDs that range from 0-50. A simple lookup table might look like the following:

The field on which the range is based is also known as the **shard key**. Naturally, the choice of shard key, as well as the ranges, are critical in making range-based sharding effective. A poor choice of shard key will lead to unbalanced shards, which leads to decreased performance. An effective shard key will allow for queries to be targeted to a minimum number of shards. In our example above, if we query for all records with IDs 10-30, then only shards A and B will need to be queried.

Two key attributes of an effective shard key are high cardinality and well-distributed frequency. Cardinality refers to the number of possible values of that key. If a shard key only has three possible values, then there can only be a maximum of three shards. Frequency refers to the distribution of the data along the possible values. If 95% of

records occur with a single shard key value, then due to this hotspot, 95% of the records will be allocated to a single shard. Consider both of these attributes when selecting a shard key.

Range-based sharding is an easy-to-understand method of horizontal partitioning, but the effectiveness of it will depend heavily on the availability of a suitable shard key and the selection of appropriate ranges. Additionally, the lookup service can become a bottleneck, although the amount of data is small enough that this typically is not an issue.

Algorithmic/Hashed Sharding

Algorithmic sharding or **hashed sharding**, takes a record as an input and applies a hash function or algorithm to it which generates an output or hash value. This output is then used to allocate each record to the appropriate shard.

The function can take any subset of values on the record as inputs. Perhaps the simplest example of a hash function is to use the modulus operator with the number of shards, as follows:

Hash Value = ID % Number of Shards

This is similar to range-based sharding—a set of fields determines the allocation of the record to a given shard. Hashing the inputs allows more even distribution across shards even when there is not a suitable shard key, and no lookup table needs to be maintained. However, there are a few drawbacks.

First, query operations for multiple records are more likely to get distributed across multiple shards. Whereas ranged sharding reflects the natural structure of the data across shards, hashed sharding typically disregards the meaning of the data. This is reflected in increased broadcast operation occurrence.

Second, resharding can be expensive. Any update to the number of shards likely requires rebalancing all shards to moving around records. It will be difficult to do this while avoiding a system outage.

Entity-/Relationship-Based Sharding

Entity-based sharding keeps related data together on a single physical shard. In a relational database (such as PostgreSQL, MySQL, or SQL Server), related data is often spread across several different tables.

For instance, consider the case of a shopping database with Users and Payment Methods. Each user has a set of payment methods that is tied tightly with that user. As such, keeping related data together on the same shard can reduce the need for broadcast operations, increasing performance.

Geography-Based Sharding

Geography-based sharding, or geosharding, also keeps related data together on a single shard, but in this case, the data is related by geography. This is essentially ranged sharding where the shard key contains geographic information and the shards themselves are geo-located.

For example, consider a dataset where each record contains a “country” field. In this case, we can both increase overall performance and decrease system latency by creating a shard for each country or region, and storing the appropriate data on that shard. This is a simple example, and there are many other ways to allocate your geoshards which are beyond the scope of this article.

Summary

We’ve defined what sharding is, when to use it, and different sharding architectures. Sharding is a great solution for applications with large data requirements and

high-volume read/write workloads, but it does come with additional complexity. Consider whether the benefits outweigh the costs or whether there is a simpler solution before you begin implementation.