Four Data Sharding Strategies We Analyzed in Building a Distributed SQL Database

Karthik Ranganathan | Founder & CTO

January 14, 2020

A distributed SQL database needs to automatically partition the data in a table and distribute it across nodes. This is known as data sharding and it can be achieved through different strategies, each with its own tradeoffs. In this post, we will examine various data sharding strategies for a distributed SQL database, analyze the tradeoffs, explain the rationale for which of these strategies YugabyteDB supports and what we picked as the default sharding strategy.

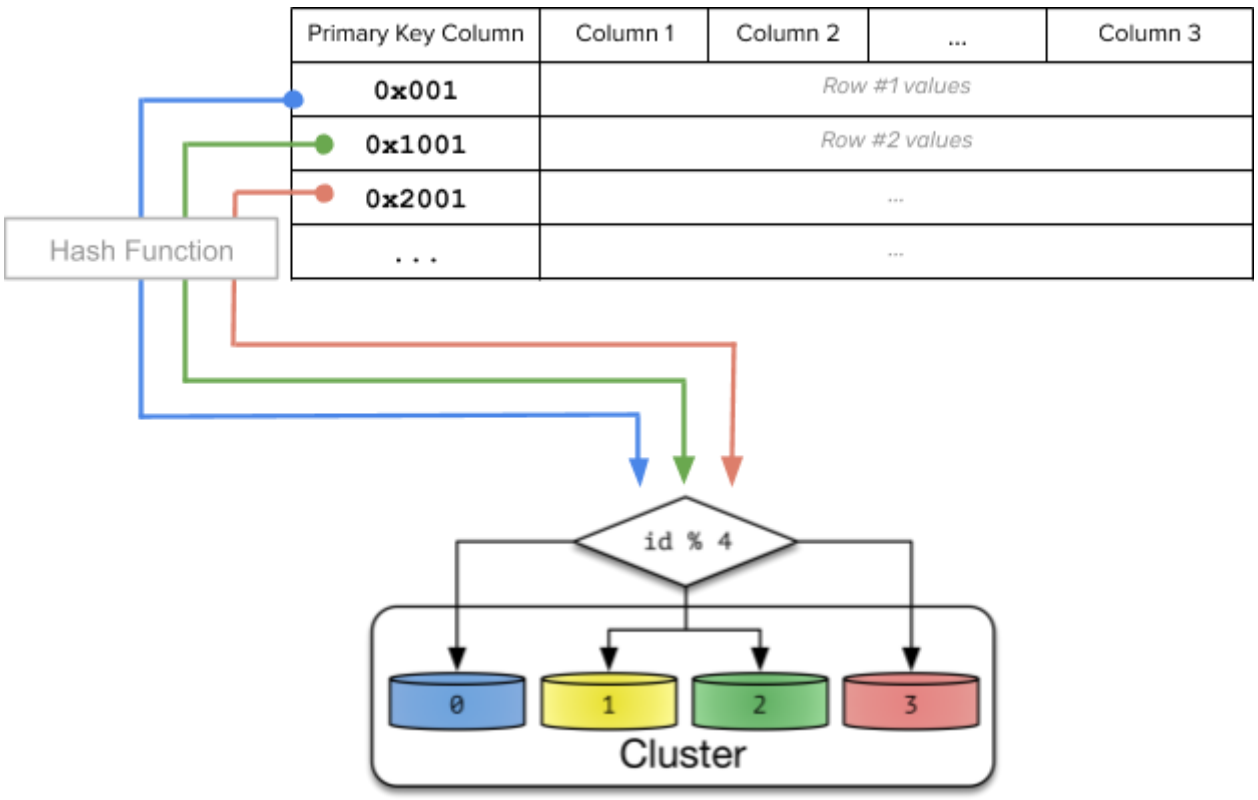
> ***What's YugabyteDB?*** *It is an open source, high-performance distributed SQL database built on a scalable and fault-tolerant design inspired by Google Spanner. Yugabyte's SQL API (YSQL) is PostgreSQL wire compatible.*

Lessons from building sharded data systems

Data sharding helps in scalability and geo-distribution by horizontally partitioning data. A SQL table is decomposed into multiple sets of rows according to a specific sharding strategy. Each of these sets of rows is called a shard. These shards are distributed across multiple server nodes (containers, VMs, bare-metal) in a shared-nothing architecture. This ensures that the shards do not get bottlenecked by the compute, storage and networking resources available at a single node. High availability is achieved by replicating each shard across multiple nodes. However, the application interacts with a SQL table as one logical unit and remains agnostic to the physical placement of the shards. In this section, we will outline the pros, cons and our practical learnings from the sharding strategies adopted by these databases.

Memcached and Redis – Algorithmic Sharding

Distributed caches have had to distribute data across multiple nodes for a while. A commonly used technique is algorithmic sharding, where each key consistently maps to the same node. This is achieved by computing a numeric hash value out of the key and computing a modulo of that hash using the total number of nodes to compute which node owns the key.



Part of the image from source: How Sharding Works

Pros

In algorithmic sharding, the client can determine a given partition's database without any help.

Cons

When a new node is added or removed, the ownership of almost all keys would be affected, resulting in a massive redistribution of all the data across nodes of the cluster. While this is not a correctness issue in a distributed cache (because cache misses will repopulate the data), it can have a huge performance impact since the entire cache will have to be warmed again.

Analysis

Adding and removing nodes is fundamental to a distributed database, and these operations need to be efficient.

This makes this type of sharding a poor option and is not implemented in YugabyteDB.

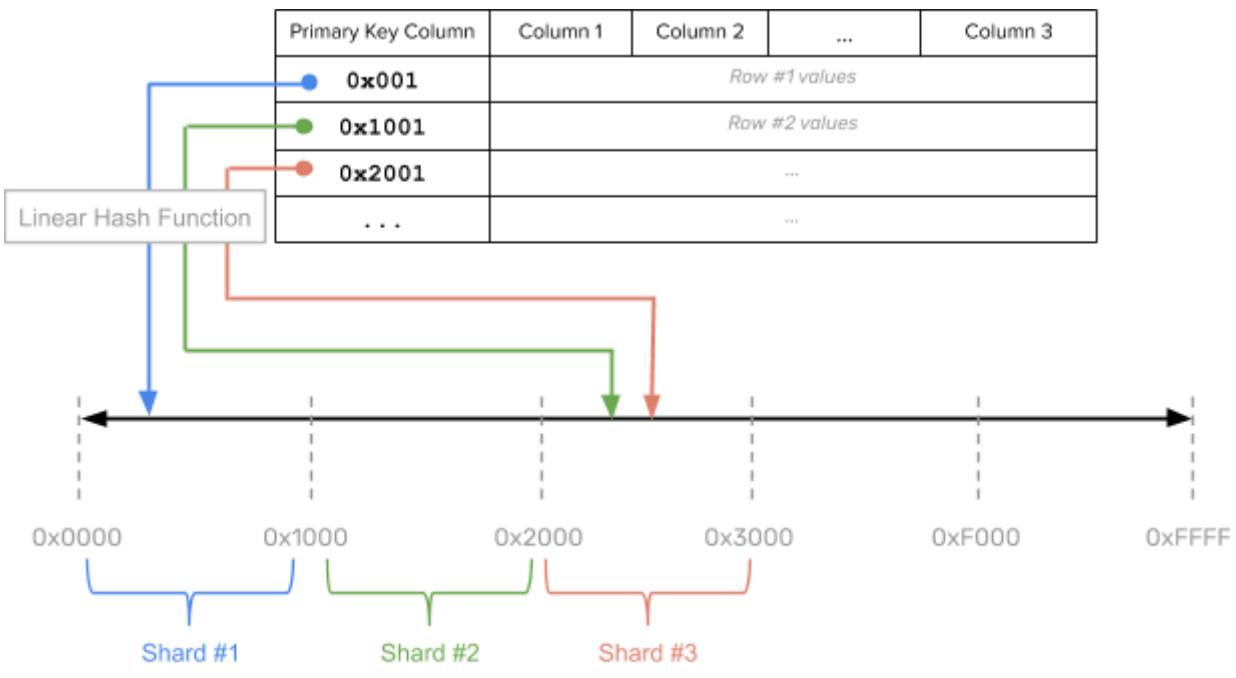
References:

- Sharding data across a memcache tier from "Distributed Caching with Memcached"
- Algorithmic sharding from "How Sharding Works"

Initial Implementation in Cassandra – Linear Hash Sharding

Various members of the Yugabyte team were directly involved in the early days of building Cassandra at Facebook (circa 2007), well before it became an open source project. While designing data sharding in Cassandra, we looked to Bigtable from Google (which did range sharding by default) and Dynamo from Amazon (which did consistent hash sharding by default). As an attempt to achieve the best of both worlds, we opted for linear hash sharding, also known as the *OrderPreservingPartitioner* in Cassandra.

Linear hash sharding is a hybrid between hash and range sharding that preserves the sort order of the rows by utilizing a linear hash function instead of a regular random hash function to compute how to shard the rows. A linear hash function, sometimes referred to as an order preserving hash, is a hash function that maintains the relative ordering of input values while changing their distribution spacing. This type of sharding preserves the sort ordering of the rows, while redistributing these rows across a larger key space. The idea is that the larger key space over which the redistribution of rows is done can be pre-sharded, enabling the table to be spread across multiple nodes.



Pros

In theory, this type of sharding allows efficiently querying a range of rows by the primary key values while enabling pre-splitting of the table into multiple shards.

Cons

In practice, this sharding strategy was problematic because it was impossible to pick good shard split boundaries ahead of time. The primary disadvantage of linear hash sharding is that the data is never

evenly distributed between partitions, and thus results in hotspots.

YugabyteDB 2.2

Analysis

While useful in theory, these are only a narrow set of use-cases that can leverage this sharding strategy effectively. In the case of Cassandra, the default sharding strategy was changed from linear hash sharding to consistent sharding to improve scalability and performance.

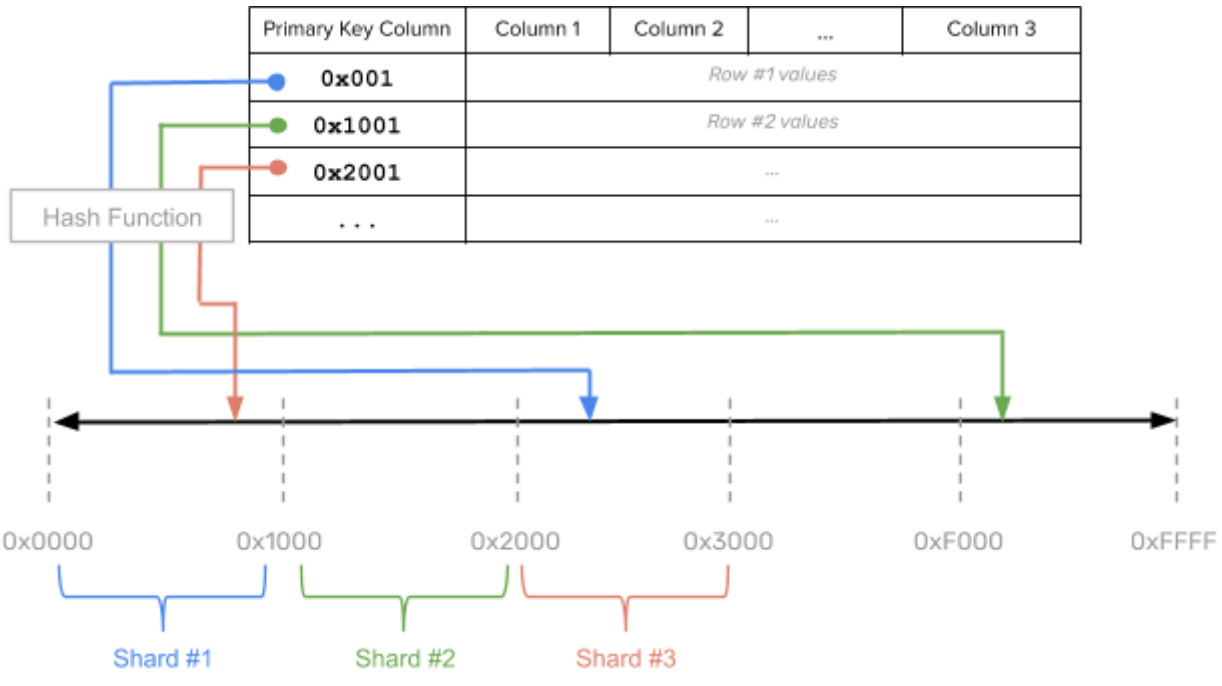
This is documented in the Apache Cassandra docs thus:

> Not recommended both because of this limitation and because globally ordering all your partitions generates hot spots: some partitions close together will get more activity than others, and the node hosting those will be overloaded relative to others. You can try to mitigate with active load balancing but this works poorly in practice; by the time you can adjust token assignments so that less hot partitions are on the overloaded node, your workload often changes enough that the hot spot is now elsewhere. Remember that preserving global order means you can't just pick and choose hot partitions to relocate, you have to relocate contiguous ranges.

Hence, this is a poor sharding strategy and is not implemented in YugabyteDB.

DynamoDB and Cassandra – Consistent Hash Sharding

With consistent hash sharding, data is evenly and randomly distributed across shards using a partitioning algorithm. Each row of the table is placed into a shard determined by computing a consistent hash on the partition column values of that row. This is shown in the figure below.



Pros

This sharding strategy is ideal for massively scalable workloads because it distributes data evenly across all the nodes in the cluster, while retaining ease of adding nodes into the cluster. Recall from earlier that algorithmic hash sharding is very effective also at distributing data across nodes, but the distribution strategy depends on the number of nodes. With consistent hash sharding, there are many more shards than the number of nodes and there is an explicit mapping table maintained tracking the assignment of shards to nodes. When adding new nodes, a subset of shards from existing nodes can be efficiently moved into the new nodes without requiring a massive data reassignment.

Cons

Performing range queries could be inefficient. Examples of range queries are finding rows greater than a lower bound or less than an upper bound (as opposed to point lookups).

Analysis

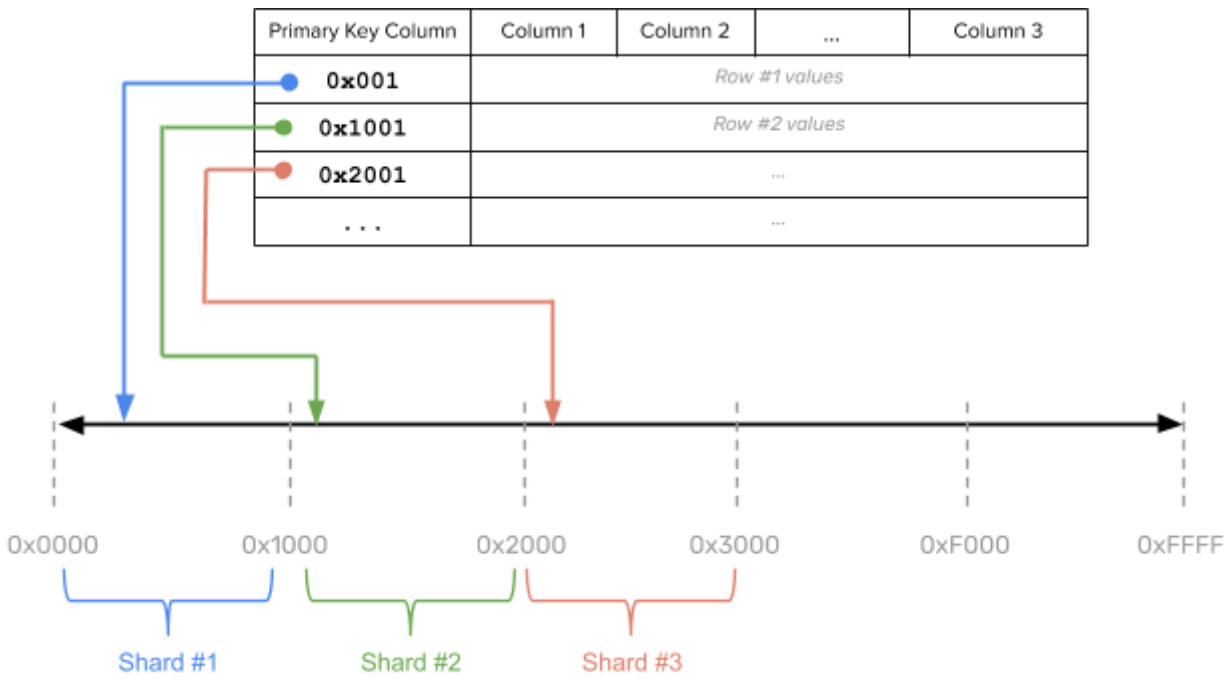
As a team with experience building and running multiple database systems, we have found hash sharding to be ideal for workloads requiring scale. For example, without any exception, all the massively scalable services we ran at Facebook on Apache HBase implemented hash sharding at the application layer and pre-split tables. Without hash sharding, these applications would have hit severe bottlenecks.

We decided YugabyteDB should support consistent hash sharding.

Google Spanner and HBase – Range Sharding

Apache HBase is a massively scalable, distributed NoSQL database modelled after Google BigTable. This is another database that many members in the Yugabyte team are familiar with given they built and ran HBase at scale inside Facebook many years ago. It was the database that backed multiple internet-scale services such as Facebook Messenger (the user to user messaging platform) and the Operational Data Store (which powered metrics and alerts across all Facebook infrastructure). HBase, as well as Google Spanner, have support for range sharding.

Range sharding involves splitting the rows of a table into contiguous ranges that respect the sort order of the table based on the primary key column values. The tables that are range sharded usually start out with a single shard. As data is inserted into the table, it is dynamically split into multiple shards because it is not always possible to know the distribution of keys in the table ahead of time. The basic idea behind range sharding is shown in the figure below.



Pros

This type of sharding allows efficiently querying a range of rows by the primary key values. Examples of such a query is to look up all keys that lie between a lower bound and an upper bound.

Cons

Range sharding leads to a number of issues in practice at scale, some of which are similar to that of linear hash sharding.

Firstly, when starting out with a single shard implies only a single node is taking all the user queries. This often results in a database “warming” problem, where all queries are handled by a single node even if there are multiple nodes in the cluster. The user would have to wait for enough splits to happen and these shards to get redistributed before all nodes in the cluster are being utilized. This can be a big issue in production workloads. This can be mitigated in some cases where the distribution is keys is known ahead of time by pre-splitting the table into multiple shards, however this is hard in practice.

Secondly, globally ordering keys across all the shards often generates hot spots: some shards will get much more activity than others, and the node hosting those will be overloaded relative to others. While these can be mitigated to some extent with active load balancing, this does not always work well in practice because by the time hot shards are redistributed across nodes, the workload could change and introduce new hot spots.

Analysis

Range sharding is essential for efficiently supporting queries looking up a range of rows based on column values that are less than, greater than or that lie between some user specified values. Subjecting such queries to a scheme that is not range sharded could be prohibitive in performance, since it might have to perform a full table scan. This makes this strategy essential.

When range sharding is picked in scenarios that do not require the primary keys to be ordered, applications run into scalability bottlenecks as mentioned in the cons section above. An often recommended workaround is to implement hash sharding on top of range sharding. But in practice, users do not always remember to implement hash sharding on top of range sharding.

Given that range sharding is useful in certain scenarios, we decided YugabyteDB should support range sharding.

Putting it all together

Hash sharding enables utilizing multiple nodes evenly from the get go, and is very effective at preventing hot spots. Range sharding is good for performing range searches (for example, key range in between two values), and is just one of many access patterns possible in a distributed database. The table below summarizes the key tradeoffs between range and hash sharding.

	Consistent Hash Sharding	Range Sharding
Supports pre-splitting (to prevent database warming problem)?	Yes	No
Can efficiently perform range scans on large datasets?	No	Yes
Prevents hotspots in the database (hence works well for massive scale)?	Yes	No

Support for consistent hash and range sharding

Because all of the above properties are desirable, we decided to include both consistent hash and range sharding in YugabyteDB. This gives the user full control over how to shard data when creating a table. Let us review these strategies through examples of YugabyteDB tables for an eCommerce application. The following examples are from the standard SQL dataset known as Northwinds.

Hash sharding is ideal for a majority of tables, such as the customers, product catalog and the shopping cart. The example below shows creating a customer table with hash sharding.

```
CREATE TABLE customers (
    customer_id bpchar NOT NULL,
    company_name character varying(40) NOT NULL,
    contact_name character varying(30),
    contact_title character varying(30),
    address character varying(60),
    city character varying(15),
    region character varying(15),
    postal_code character varying(10),
    country character varying(15),
    phone character varying(24),
    fax character varying(24),
    PRIMARY KEY (customer_id HASH)
);
```

Now, let us say we want to perform range queries on the orders table (they way it is defined in this dataset), and therefore want to range shard it. This can be done as follows:

```
CREATE TABLE order_details (
    order_id smallint NOT NULL,
    product_id smallint NOT NULL,
    unit_price real NOT NULL,
    quantity smallint NOT NULL,
    discount real NOT NULL,
    PRIMARY KEY (order_id ASC, product_id),
    FOREIGN KEY (product_id) REFERENCES products,
    FOREIGN KEY (order_id) REFERENCES orders
);
```

Consistent hash as the default sharding strategy

The next question was how to pick the default sharding strategy. Defaults are meant to serve the target use case. Users turn to YugabyteDB primarily for scalability reasons. Most use cases requiring scalability often do not need to perform range lookups on the primary key, hence we picked consistent hash sharding as the default in YugabyteDB. User identity (user ids do not need ordering), product catalog (product ids are not related to one another) and stock ticker data (one stock symbol is independent of all other stock symbol) are some common examples.

In use cases when range lookups are desired, YugabyteDB allows specifying range sharding, along with optionally pre-splitting the data. In cases when range queries become important after the data is already loaded into a hash sharded table, a range sharded secondary index can be created on that column. Once the secondary index is rebuilt, range queries would become efficient.

Future work

Note that automatic tablet splitting is an essential feature to enable continuous sharding of data, especially for range sharding. Automated tablet splitting was included in YugabyteDB releases starting in version 2.2, and work continues to make this feature generally available and enabled by default, currently scheduled for the upcoming 2.4 release. Visit the planned roadmap to follow along, and/or please subscribe to this GitHub issue.

Conclusion

Consistent hash and range sharding are the most useful data sharding strategies for a distributed SQL database. Consistent hash sharding is better for scalability and preventing hot spots, while range sharding is better for range based queries. YugabyteDB supports both hash and range sharding of data across nodes to enable the best of both worlds, with hash sharding as the default.

What's Next?

- Compare YugabyteDB in depth to databases like CockroachDB, Google Cloud Spanner and MongoDB.
- Get started with YugabyteDB on macOS, Linux, Docker, and Kubernetes.
- Contact us to learn more about licensing, pricing or to schedule a technical overview.

JANUARY 14, 2020

CLOUD NATIVE , DISTRIBUTED SQL , PARTITIONING , POSTGRESQL

Karthik Ranganathan | Founder & CTO
January 14, 2020

Related Posts