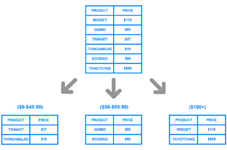
System Design — Sharding / Data Partitioning

Peng Yang | Follow
Apr 4 · 4 min read



References

- Understanding database sharding

Table of Contents

1. Partitioning method

1. 1 Horizontal partitioning — also known as sharding

It is a range-based sharding. You put different rows into different tables, the structure of the original table stays the same in the new tables, i.e., we have the same number of columns.

- When partitioning your data, you need to assess the number of rows in the new tables, so each table has the same number of data and will grow by a similar number of new customers in the future.
- Con: If the value whose range is used for sharding isn't chosen carefully, the partitioning scheme will lead to unbalanced servers.

1. 2 Vertical partitioning

This type of partition divides the table vertically (by columns), which means that the structure of the main table changes in the new ones.

- An ideal scenario for this type of partition is when you don't need all the information about the customer in your query.
- **Divide data for a specific feature to their own server**. When you have different types of data in your database, such as names, dates, and pictures. You could keep the string values in SQL DB (expensive), and pictures in an Azure Blob (cheap).
- Pro: Straightforward to implement. Low impact on the application.
- Con: To support the growth of the application, a database may need further partitioning.

1.3 Directory-based partitioning

- A lookup service that knows the partitioning scheme and abstracts it away from the database access code.
- Allow the addition of DB servers or change of partitioning schema without impacting the application.
- Con: Can be a single point of failure.

2. Partitioning criteria

Link: https://dev.mysql.com/doc/mysql-partitioning-excerpt/5.7/en/partitioning-types.html

2.1 Range partitioning

- This type of partitioning assigns rows to partitions based on column values falling within a given range. e.g, store_id is a column of table.

```
PARTITION BY RANGE (store_id) (
    PARTITION p0 VALUES LESS THAN (6),
    PARTITION p1 VALUES LESS THAN (11),
    PARTITION p2 VALUES LESS THAN (16),
    PARTITION p3 VALUES LESS THAN (21)
);
```

2.2 Key or hash-based partitioning

- Apply a hash function to some key attribute of the entry to get the partition number. e.g. partition based on the year in which an employee was hired.

```
CREATE TABLE employees (
    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30),
    hired DATE NOT NULL DEFAULT '1970-01-01',
    separated DATE NOT NULL DEFAULT '9999-12-31',
    job_code INT,
    store_id INT
)
PARTITION BY HASH( YEAR(hired) )
PARTITIONS 4;
```

- Problem
  1. Adding new servers may require changing the hash function, which would need a redistribution of data and downtime for the service.
  2. Workaround: consistent hashing.

2.3 List partitioning

- Similar to partitioning by `RANGE`, except that the partition is selected based on columns matching one of a set of discrete values. Each partition is assigned a list of values. e.g.

```
PARTITION BY LIST(store_id) (
    PARTITION pNorth VALUES IN (3,5,6,9,17),
    PARTITION pEast VALUES IN (1,2,10,11,19,20),
    PARTITION pWest VALUES IN (4,12,13,14,18),
    PARTITION pCentral VALUES IN (7,8,15,16)
);
```

2.4 Round-robin partitioning

- With `n` partitions, the `i` tuple is assigned to partition `i % n`.

2.5 Composite partitioning

- Combine any of the above partitioning schemes to devise a new scheme.
- Consistent hashing is a composite of hash and list partitioning.
- Key -> reduced key space through hash -> list -> partition.

3. Common problems of sharding

Most of the constraints are due to the fact that operations across multiple tables or multiple rows in the same table will no longer run on the same server.

3.1 Joins and denormalization

- Joins will not be performance efficient since data has to be compiled from multiple servers.
- Workaround: **denormalize the database so that queries can be performed from a single table**. But this can lead to data inconsistency.

3.2 Referential integrity

- Difficult to enforce data integrity constraints (e.g. foreign keys).
- Workaround
  1. Referential integrity is enforced by application code.
  2. Applications can run SQL jobs to clean up dangling references.

3.3 Rebalancing

- Necessity of rebalancing
  1. Data distribution is not uniform.
  2. A lot of load on one shard.
- Create more DB shards or rebalance existing shards changes partitioning scheme and requires data movement.

Thank you for reading! if you liked this blog, please also check my other blogs of System Design series.

- System Design — Load Balancing
- System Design — Caching
- System Design — Sharding / Data Partitioning
- System Design — Indexes
- System Design — Proxies
- System Design — Message Queues
- System Design — Redundancy and Replication
- System Design — SQL vs. NoSQL
- System Design — CAP Problem
- System Design — Consistent Hashing
- System Design — Client-Server Communication
- System Design — Storage
- System Design — Other Topics
- Object-Oriented Programming — Basic Design Patterns in C++

System Design Interview | Sharding

WRITTEN BY
Peng Yang | Follow
Software Infrastructure Engineer, Aim to become an engineer who understands computer science very well.
https://www.linkedin.com/in/peng-larry-yang-9a794561/