Grokking the System Design Interview
Search

- System Design Basics
  - Why System Design Interviews?
  - System Design Basics
  - Load Balancing
  - Caching
  - Sharding or Data Partitioning
  - Indexes
  - Proxies
  - Queues
  - Redundancy and Replication
  - SQL vs. NoSQL
  - CAP Theorem
  - Consistent Hashing
  - Long-Polling vs WebSockets vs Server-Sent Events (*New*)
- System Design Problems
  - System Design Interviews: A step by step guide
  - Designing a URL Shortening service like TinyURL
  - Designing Pastebin
  - Designing Instagram
  - Designing Dropbox
  - Designing Facebook Messenger
  - Designing Twitter
  - Designing Youtube or Netflix
  - Designing Typeahead Suggestion
  - Designing an API Rate Limiter (*New*)
  - Designing Twitter Search
  - Designing a Web Crawler
  - Designing Facebook's Newsfeed
  - Designing Yelp or Nearby Friends
  - Designing Uber backend
  - Design BookMyShow (*New*)
- Contact Us
  - Feedback

LearnTeach

- My Profile
  - View
  - Edit
- Logout

Sharding or Data Partitioning

Data partitioning (also known as sharding) is a technique to break up a big database (DB) into many smaller parts. It is the process of splitting up a DB/table across multiple machines to improve the manageability, performance, availability and load balancing of an application. The justification for data sharding is that, after a certain scale point, it is cheaper and more feasible to scale horizontally by adding more machines than to grow it vertically by adding beefier servers.

1. Partitioning Methods

There are many different schemes one could use to decide how to break up an application database into multiple smaller DBs. Below are three of the most popular schemes used by various large scale applications.

a. Horizontal partitioning: In this scheme, we put different rows into different tables. For example, if we are storing different places in a table, we can decide that locations with ZIP codes less than 10000 are stored in one table, and places with ZIP codes greater than 10000 are stored in a separate table. This is also called a range based sharding, as we are storing different ranges of data in separate tables.

The key problem with this approach is that if the value whose range is used for sharding isn't chosen carefully, then the partitioning scheme will lead to unbalanced servers. In the previous example, splitting location based on their zip codes assumes that places will be evenly distributed across the different zip codes. This assumption is not valid as there will be a lot of places in a thickly populated area like Manhattan compared to its suburb cities.

b. Vertical Partitioning: In this scheme, we divide our data to store tables related to a specific feature to their own server. For example, if we are building Instagram like application, where we need to store data related to users, all the photos they upload and people they follow, we can decide to place user profile information on one DB server, friend lists on another and photos on a third server.

Vertical partitioning is straightforward to implement and has a low impact on the application. The main problem with this approach is that if our application experiences additional growth, then it may be necessary to further partition a feature specific DB across various servers (e.g. it would not be possible for a single server to handle all the metadata queries for 10 billion photos by 140 million users).

c. Directory Based Partitioning: A loosely coupled approach to work around issues mentioned in above schemes is to create a lookup service which knows your current partitioning scheme and abstracts it away from the DB access code. So, to find out where does a particular data entity resides, we query our directory server that holds the mapping between each tuple key to its DB server. This loosely coupled approach means we can perform tasks like adding servers to the DB pool or change our partitioning scheme without having to impact your application.

2. Partitioning Criteria

a. Key or Hash-based partitioning: Under this scheme, we apply a hash function to some key attribute of the entity we are storing, that yields the partition number. For example, if we have 100 DB servers and our ID is a numeric value that gets incremented by one, each time a new record is inserted. In this example, the hash function could be 'ID % 100', which will give us the server number where we can store/read that record. This approach should ensure a uniform allocation of data among servers. The fundamental problem with this approach is that it effectively fixes the total number of DB servers, since adding new servers means changing the hash function which would require redistribution of data and downtime for the service. A workaround for this problem is to use Consistent Hashing.

b. List partitioning: In this scheme, each partition is assigned a list of values, so whenever we want to insert a new record, we will see which partition contains our key and then store it there. For example, we can decide all users living in Iceland, Norway, Sweden, Finland or Denmark will be stored in a partition for the Nordic countries.

c. Round-robin partitioning: This is a very simple strategy that ensures uniform data distribution. With 'n' partitions, the 'i' tuple is assigned to partition (i mod n).

d. Composite partitioning: Under this scheme, we combine any of above partitioning schemes to devise a new scheme. For example, first applying a list partitioning and then a hash based partitioning. Consistent hashing could be considered a composite of hash and list partitioning where the hash reduces the key space to a size that can be listed.

3. Common Problems of Sharding

On a sharded database, there are certain extra constraints on the different operations that can be performed. Most of these 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. Below are some of the constraints and additional complexities introduced by sharding:

a. Joins and Denormalization: Performing joins on a database which is running on one server is straightforward, but once a database is partitioned and spread across multiple machines it is often not feasible to perform joins that span database shards. Such joins will not be performance efficient since data has to be compiled from multiple servers. A common workaround for this problem is to denormalize the database so that queries that previously required joins can be performed from a single table. Of course, the service now has to deal with all the perils of denormalization such as data inconsistency.

b. Referential integrity: As we saw that performing a cross-shard query on a partitioned database is not feasible, similarly trying to enforce data integrity constraints such as foreign keys in a sharded database can be extremely difficult.

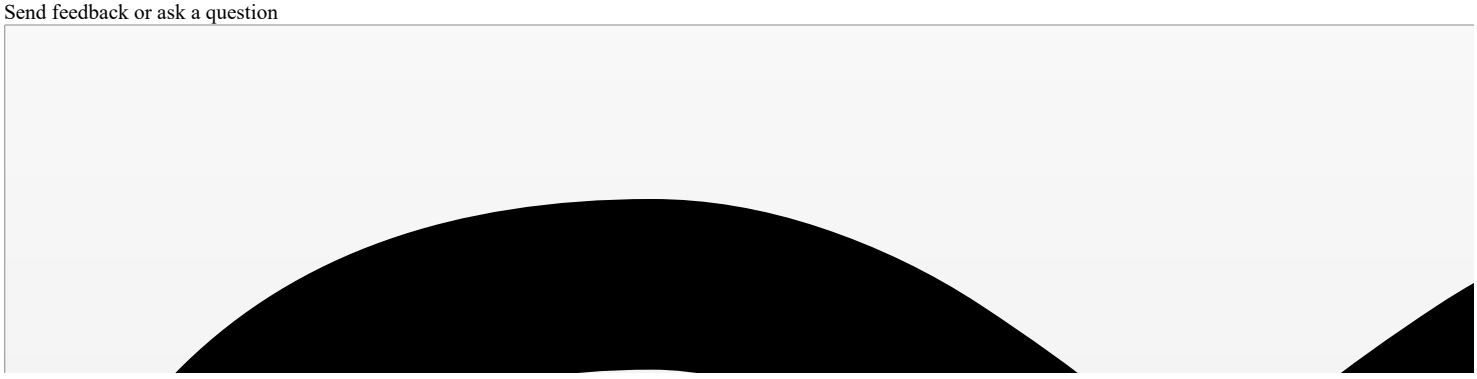
Most of RDBMS do not support foreign keys constraints across databases on different database servers. Which means that applications that require referential integrity on sharded databases often have to enforce it in application code. Often in such cases, applications have to run regular SQL jobs to clean up dangling references.

c. Rebalancing: There could be many reasons we have to change our sharding scheme:

1. The data distribution is not uniform, e.g., there are a lot of places for a particular ZIP code, that cannot fit into one database partition.
2. There are a lot of load on a shard, e.g., there are too many requests being handled by the DB shard dedicated to user photos.

In such cases, either we have to create more DB shards or have to rebalance existing shards, which means the partitioning scheme changed and all existing data moved to new locations. Doing this without incurring downtime is extremely difficult. Using a scheme like directory based partitioning does make rebalancing a more palatable experience at the cost of increasing the complexity of the system and creating a new single point of failure (i.e. the lookup service/database).

Mark as completed
← *Previous*Caching*Next* →Indexes
Send feedback or ask a question



25 recommendations