DB

Background

NoSQL has been around claiming better performance and distributed scalability for quite a long time now. SQL database performance increased during the same period of time especially with wide spread adoption of fast storages like SSDs. NoSQL has limitations. Its performance is not exceeding RDBMs with several benchmarks. Its CAP model does not support transaction. Cluster boot time can be very long and re-hashing on a node faulre takes some time.

Stability is the core of backend service and stability of persistent data management is one of the most important factors. RDBMS has always been a winner after all the new technologies. What would be the future? We do not know it exactly, but RDMBS will be one of the key players for quite long.

lax will use RDBMS as its main data management techonology. Hash based or key based sharding will be added to support large service covering billion users.

Sharding

A simple idea of partitioned world has been used in games. Games have characters as the main data of a game. Character is the key to other data like item, inventory, money and other game assets. Manual sharding can be provided with a character key like character name or a field directing the database for a character.

Shard function

A function $shard : C \rightarrow DB$ can be used to target the database to create, update, select data from. The $shard$function can be based on a field in account database.

Transaction between DBs

Two DBs need to interact sometimes when trading between users, or giving items to other user. These transactions can be provided in application layer with two phase commit algorithm.

Another technique commonly used is to have a separate DB for long term transactions. A typical example is to have mailbox db to give an item and recieve that item from another user.

Aggregation DB and Aggregation Services

Some containers range over shard DBs. Example is the guild of characters on different databases. Like mailbox DB, guild table can be created on aggreate DB. This table can be loaded to one guild_service providing guild aggregation service. Operations of guild_service is to apply, accept, kickout, chat, and so on.

User DB and User Service

To enter a service or a game, users need to get authenticated, i.e., login into the service. User DB is used to store account and password to check the user. When facebook or google service is used to authenticate, the service needs to map the account with internal id to use the account.

To scale out, a hash based sharding can be used based on account id from user. This is the simple scheme, but can be efficient.

When a user moves around services, it needs to authorize itself with login information. user_service is the last fallback service to get that information if any other way is not possible to authenticate itself.

user_service can be also used to identify the service of a user where direct communication with the user is possible. This last fallback service of user_service can make the overall implementation simple.

user_service can use the same hash algorithm used for database sharding to scale out for huge amount of users. This hashed distribution can make the implementation more complicated, but this scheme is very powerful when implemented clear and stable. It is the core of lax clustering that supports billions of users. It is a aim, but will be tried hard.

Locational transparancy of db_service

Since RDBMS and sharding are chosen for stability and scalability, db_services that provides query service need to scale well with DBs. Local db_service on each node is convenient to impelement and manage. But when there are thousands of DBs, the number of connections to each DB can be a bottleneck. To solve this, db-service needs to be transparant in location.

db_service can be created and distributed based on shards that the service provides. For example, shard 1 / 3 on db_service 1 and shard 2 / 4 on db_service 2 supposing 4 db_services in the service.

Requirements

- use protocol messages to communicate
- select db_service with shard (or hash)
- handle failure of db_service

Design

- query
  - is a bits_message
  - has members to execute a query and process the result
- query execution
  - task_scheduler like
- locating db_service
  - use service location with role and hash
- sending to db_service
- handle response from db_service
- 2 phase commit / rollback
- failure handling
- re-hashing
  - important!!! intersting.
  - adding or removing DBs
  - automated
    - small range partition
    - make it fast with re-hash