Database Design

Problem Statement

- Test1 App Description: A user can follow multiple users in the app. Each user can post the pictures take by them. After opening the app each user will be able to see the pictures posted by the people the user is following. So, for a new user the feed will be empty, but after he starts following users, his feed will become interesting. A user can share a picture or like a picture in his feed.
  1. Design the database for the app, keeping in mind, there can be millions of users, pictures, shares and likes.
  2. From the database you have designed, find the time complexity to populate the feed. The feed will have latest 10 photos from the people the user is following.
  3. In the question no.2 we are getting the latest 10 photos, what are the other ways to populate the feed, so that it can be more interesting.
  4. What is the time complexity to get the total no.of likes and no.of shares for all the pictures that a user has posted.

Database Design

- Before designing the database we should know the data we need to retrieve in an efficient manner.
  - Get the user feed
  - Get the users followers
  - Get the users following

Data retrieval – Get the user feed

- Subproblems
  - Get the user info
  - Get the images shared by users followed
  - Get the info on like and shares on each image

SubProblem 1: Getting the user Info

- You need a database for each user where you need to save the personal info of the user
- Each user has to be identified by a unique id(primary key) which I used to retrieve the user info from the database
- A simple schema would be
  - Users{
    user_id PRIMARY KEY NOT NULL INT,
    Name TEXT,
    /* other user info */
    }
- Constraint – there are million users

Million Users Problem

- If there are million users – you cannot save the whole database in one machine. Even if you can, it would make information retrieval inefficient(would be clear in moment).
- The solution is you can partition the database to several servers.
- The technique we can use is sharding.
- http://en.wikipedia.org/wiki/Shard_(database_architecture)
- Sharding partitions the database horizontally which means by rows instead of columns(normalizations)

Advantages of partitioning

- Fast Information retrieval:
  - Ideally the database is distributed uniformally among the logical shards(servers). If you know the logical shard, you have less number of rows to search for a particular Id as compared to searching the whole database.
- Uniform Database Distribution:
  - You have multiple database instances to store for an application. You cannot save all database instances on a single machine with million rows.
  - Relationships among database instances causes a query to retrieve information from multiple instances. AS such if you have partitioned them, we can make sure all dependent pieces of information are present on a single shard(server) to make efficient retrieval.
- The load can be spread out over multiple machines, greatly improving performance

Method Used for Sharding

- No change in schema
- We need a logic to assign a row to a logical shard and retrieve it when required.
- Since each row is identified by its id, apply some logic on Ids.
- The typical solution that works for a single database—just using a database's natural auto-incrementing primary key feature—no longer works when data is being inserted into many databases at the same time.
- Instagram Link – this link proposes several solutions.
- I used the solution proposed in the link

- Each of IDs consists of:
  - 41 bits for time in milliseconds (gives us 41 years of IDs with a custom epoch)
  - 13 bits that represent the logical shard ID
  - 10 bits that represent an auto-incrementing sequence, modulus 1024. This means we can generate 1024 IDs, per shard, per millisecond
- We could have saved timestamp as a separate column but this method reduces the memory usage.
- Helps to sort the users by timestamp – feature helpful to show followers and following

Subproblem 2: get the Images

- Retrieve image by latest first
- Apply the same logic for storing image_id as user_id
- Typical Schema:

```
Images(
 image_id PRIMARY KEY NOT NULL INT,
 user_id   NOT NULL INT, // id of user who uploaded the image
 Name TEXT,
 /*other relevant info*/
     user_id references Users(user_id)
 )
```

Subproblem 3:Get the likes and shares

- likes{
  id PRIMARY KEY NOT NULL INT,
  photo_id NOT NULL INT,// photo liked
  user_id NOT NULL INT,//user who liked the photo
  photo_id references Image(image_id),
  user_id references User(user_id)
  }
- shares{
  id PRIMARY KEY NOT NULL INT,
  photo_id NOT NULL INT,//photo shared
  user_id NOT NULL INT,//user who shared the photo
  photo_id references Image(image_id)
  user_id references User(user_id)
  }

- The id contains info about timestamp(first 41 bits)
- Both the tables can be partitioned by photo_id – the more relevant query is getting likes on a photo.
- We can also maintain a copy of the table partitioned by user_id. This data is useful if we want to populate the recent activities of a user and total likes/shares for that user.

List of followers and following

- Followers{
  source_id NOT NULL INT,
  destination_id NOT NULL INT,
  created_timestamp NOT NULL INT,
  End_timestamp INT,
  PRIMARY KEY(source_id,destination_id)
  source_id references Users(user_id)
  destination_id references Users(user_id)
  }
- End_timestamp– the timestamp when user(soure_id) stops following user(destination_id), else empty
- We need to efficiently retrieve the list of followers as well as list of following
- Twitter Database Perspective Link – this link basically discusses the way to save the social graph(followers and following)
- A solution for this is to **maintain two tables** – *followers* and *following*. In this solution, each table is maintained according to the directions of the source and destination, and is partitioned by user_id.
- The following table is partitioned by source_id to retrieve the list of users followed by user(source_id) efficiently.
- The followers table is partitioned by destination_id to retrieve the list of followers of user(destination_id) efficiently.

Complexity of Data retrieval

Getting the User Info

- We want to fetch the user info using its user_id
- The algorithm of saving the user_id is discussed earlier
- We can retrieve following info from the user_id
  - The timestamp for user_creation – first 41 bits
  - Shard number in which it is stored – next 13 bits
- The shard number is really important because it really reduces the complexity of searching all the shards for a user info.
- Considering each shard has about a 1,000 users(considering 1000 shards and 1 million users). It’s a lot easier to search a row out of 1000 rows.
- In order to optimize the search further, we can try various techniques of memory read – like hashing. These are already implemented in various shard configurations managers like zookeeper.
- Typical searching in a table of sorted rows would take O(logn) time complexity. Using hashing it can be reduced to O(1) best case.

Getting the User Feed

- This task can be divided into
  - getting the users followed and
  - top N(typically N=10) photos of these users sorted by time.
- Typical Query – SELECT * FROM images
  WHERE user_id IN
  (SELECT destination_id FROM following
  WHERE source_id = given_user_id)
  ORDER BY created_timestamp DESC
  LIMIT 10;
-

- Typically partitioning helps in retrieving all the users followed because they are present on a single logical shard.
- We can get the shard number from the user_id.(mid 13 bits)
- Shard address lookup can be done in O(1) average case.
- For multiple servers, we can get data in batches executed in parallel threads. For example for example the data is distributed over 100 physical server. We can spawn 100 parallel tasks for getting users from each server.

Getting Top N photos sorted by timestamp

- After retrieving the users followed in batches. We can retrieve top k images from each batch by using bucket sort. We can create buckets on basis of timestamp such that top bucket has the latest photos.
- After getting top k images from each batch. We need to get overall top k images. We can use bucket sort again. At last when the top bucket has more than k images, we can just sort them by using quicksort/mergesort.
- The time complexity depends on the size of each batch – considering each batch has 1000 users. After getting the 1000 users we retrieve top k*1000 images. We retrieve top k(maybe >k depends on size of top bucket) images from these images by using bucket sort. Bucket Sort would typically take O(N)- where N = 1000*k here.
- Suppose there are 1000 batches – we retrieve top k images from each batch. We have top 1000*k images again. We again use bucket sort and get top k(or >k) images in O(N) time. WE now sort the last k elements by time using quicksort/mergesort in O(NlogN) time where N=k

Getting the likes and shares

- The more relevant query is getting the number of likes and shares for each photo in the feed.
- We generally show on the feed about 5 recent users(name) who have liked the photo(facebook, instagram).
- We generally don’t show the users who shared the photo(facebook) on the feed.
- Getting the number of likes is simple:
  - Select Count(*) from likes where photo_id = given_id;
  - Similar for shares
- Getting top recent 5 users is also easy – use bucket sort similar for getting top N followers.

Getting the Followers/Following

- There is always a catch – the page which lists the users followed shows at most 25 recent users on one view.
- We don’t really need to retrieve all the users followed at a time.
- We retrieve only top 25 users sorted by timestamp of following.
- Again the same bucket sort logic can be applied.
- Caching can be used to retrieve the next 25 users in advance, there is high probability that the user will want to see more users followed.(more on caching later)
- Similar for retrieving users following current user

Caching

- There are several APIs which helps in memory caching.
- One of them which is popularly used is Redis.
- In redis everthing is stored as a key-value pair.
- We can implement our redis cache similar as twitter
- Refer: http://redis.io/topics/twitter-clone

Redis Data Model for the problem

- Each user is given a media ID which essentially is a one to one mapping from media ID to redis instance.
- We can use the redis increment feature to assign media IDs
- The users are stored as a hash
- For Example:
  - INCR next_user_id => 1000
  - HMSET user:1000 username antirez password p1pp0
- We also require a reverse mapping of username to media ID
  - HSET users antirez 1000
- The reverse mapping is required during login, the user enters his username and password, we should know for a username, what is the media ID and using that media ID, we can authenticate the password.
- We also need an internal mapping of media ID to user_id in database to retrieve data from the database(cache miss).

Storing and retrieving Followers/Following

- We store followers/following as sorted set
- followers:1000 => Sorted Set of media IDs of all the followers users of user 1000.
- following:1000 => Sorted Set of media IDs of all the following users of user 1000.
- They are sorted by timestamp of following
- Suppose a user 1000 follows user 5000
- We need to make two entries:
  - ZADD following:1000 1401267618 5000
  - ZADD followers:5000 1401267618 1000 => 1401267618 is the timestamp
- Helps in efficient retrieval of both followers and following

Storing and retrieving Images for feed

- We store posts(images) as a list where every new entry is left pushed
- So everytime a user refreshes the page to get new posts, we use LRANGE to get the list of new postIDs(image IDs) limited by a count. The count is the number of images that can be shown on a single page.
- For each Image ID, we retrieve the images and other metadata to be rendered as html to be shown to the user.
- If the number of posts goes to millions, we can resort to Sorted Set for more efficient retrieval of updates. However, its highly likely that a user will navigate to the millionth post.

Fanout-On-Write approach – efficient feed population

- rely on asynchronous tasks to populate individual feeds as photos are posted.
- Each time a photo is posted, the system finds out all the users followers, and assigns individual tasks to place the photo into each followers feed.
- very well suited for fast reads.
- More on this: Vmware Blog

- O(1) read cost
- O(N) write cost – N = number of followers
- Reads outweigh writes
- Justin Bieber has 7 million followers(unfortunately :P)
- Writes have to be done asynchronously
- A message Broker(celery) buffers the tasks
- Task manager(RabbitMQ) distributes the tasks to workers
- Multiple parallel tasks are assigned to workers working asynchronously.
- Reassignment of task to different worker if a worker fails.

Ways to populate Feed

- The most common feed contains latest photos of the users one has followed
- How can the feed be made more interesting – the parameters
  - Number of likes and shares – more the number of likes and shares from followers higher on the feed
  - User Interests – if a user likes photos of pets, a feed containing more photos of pets would make sense.
  - Geo-location – if a user is from India, photos showcasing India would make sense.
  - Relevance - By showcasing photos that friends of yours have liked, you have the opportunity to see new photographers you may not have come across. Seeing posts that your friends like is great for exposing you to new users — friends of friends. Because as we know from Facebook, people become friends with people similar to them in age, demographic and interest so you are more likely to follow a friend of a friend
  - Images that reference a trending topic

- Photos that receive a high volume of likes, comments, or shares in a short time
- Photos from users which have more number of common followers
- Photos reported by other users or flagged as spam by other users should not be present on feed
- Photos already seen/liked/shared by the user should have low visibility.

Total Number of Likes/Shares For User(Not in cache)

- To get this data efficiently we should have the likes/shares table partitioned by user_id.
- Disadvantages – replication of data
- If we don't intend to include this metric on the user feed, we don't need to do this in an efficient manner thus avoiding replication. This can be done in the background and using redis cache.
- Without partitioning by user_id, worst case it has to traverse the whole database to count the likes for a user. Even if we do this in parallel workers, it will be O(N) where N is the number of rows processed in a batch.
- With partitioning, we really need to know the start and end address of the logical shard which stores likes for a user_id. I am assuming the start and end addresses are stored in a hashtable hashed by user_id. As such the number of likes involves calculating the number of rows = (end address-start address)/size of row. Complexity – O(1)

Total Number of Likes/Shares(Using redis cache)

- Considering unpartitioned Database by user_id
- We really don’t need to query the number of likes at the beginning of the user session.
- Considering the total number of likes/shares are shown on a separate page as the user feed. We get a window of few seconds atleast before the user decides to navigate to this page.
- We can initiate multiple worker asynchronous tasks to retrieve the total number of likes and shares for the user. This would take max few seconds.
- We can store this info as a simple key-value pair on redis server.
- For Ex – SET TotalLikes:1000 500000 => 1000 is the media ID of the user
- Similar for shares
- While in the cache if a user likes/shares photo of another user – we increment the value of total likes/total shares for that user in the cache and retrieve this when required.
- If we show the total likes/total shares as a part of user feed, we have no choice but to partition the database by user_id to retrieve the metric by the time other data is retrieved(few milliseconds).

Other Challenges

- If a user start following another user – how do you update the feed.
  - We need to merge the current feed with the posts of the new user followed
- If a user unfollows another user – how do you update the feed.
  - We need to subtract the posts of the user unfollowed from the current feed
- Search functionality – photo/user
- Failure of redis instances – master – slave configuration. Switch master and slave and then replicate the master to slave.
- Managing duplicate requests – the operations should be idempotent – repeating them should have same effect.
- Managing out of order operations – the operations should be commutative. Changing the order of operation should not change the end result.
- Slow workers increases latency – kill them if reaches a threshold execution time
- Task completion acknowledgement – worker sends an acknowledgement for task completion and then fails. Task actually completes but worker sends a failure status.
- Suggestions when a user has not followed anyone – ask them for interests, show most followed users, get facebook/twitter friends

- Problem of data consistency – data in cache and disk(all copies) should be consistent.