NoSQL Contd.

Agenda

Problem Discussion

In the last class, we discussed a problem statement to design a manual sharding system which supports:

- Addition and removal of machines
- Ensures even distribution of load and storage
- Maintain configuration settings such as replication level

You can assume that the system accepts the Sharding Key and Replication level as a config input.

Sharding Key: **First letter of username**

Not a good option due to:W

- Uneven distribution of storage and load
    - there may be more usernames starting with ‘a’ than ‘x’.
- Under-utilization of resources
- Upper limit on the number of possible shards
    - There cannot be more than 26 shards in such a system
    - What if the website or application became really popular?
- If you proceed forward with an estimate of usernames with a particular letter, those estimates may not be fully accurate
- If the number of usernames starting with ‘a’ becomes exceedingly large, there is no way to shard further.

How to create a system which maintains the replication level (let’s say 3) automatically without you having to intervene?

Volunteer’s input

If the load on a machine goes beyond a certain threshold, add a new machine and transfer some part of the data to it.

Normal Drawbacks:

- Letting the sharding happen at any point in time is not desirable. What if the sharding happens at peak time? It will further degrade the response time.
- Typically, for any application, there is a traffic pattern that shows the frequency of user visits against time. It is generally seen that at a particular time, the amount of traffic is maximum, and this time is called peak time.
- In the system speculated above, it is highly likely that the data segregation to another shard will happen at peak time when the load exceeds the threshold.
- If you are segregating the machine, then you are further adding to the load. ***Because not only now do you need to handle the peak traffic, but you also migrate data at the peak traffic time.***

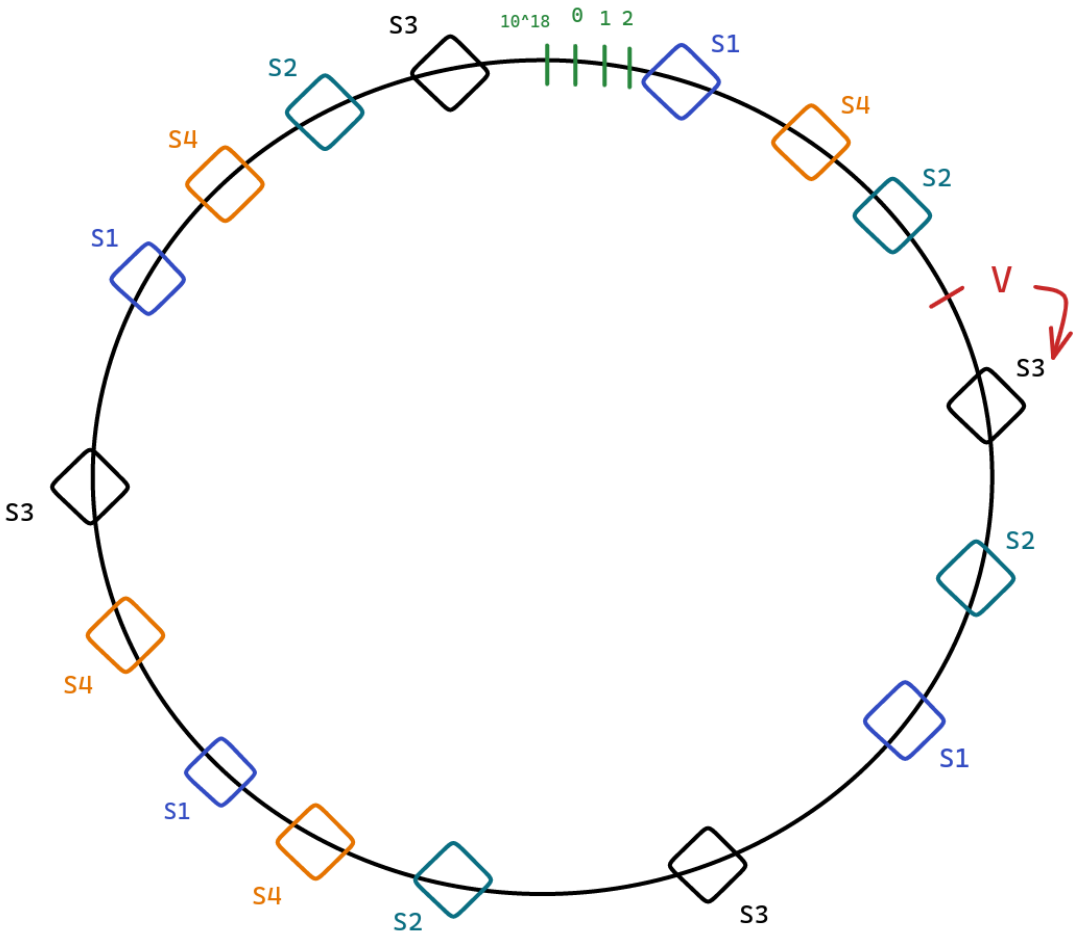
Consistent Hashing Recap

- Imagine you have three shards (S1, S2 and S3) and four different hashing functions (H1, H2, H3, H4) which produce output in the range [0, 10^18].
- Determine a unique key for each shard. For example, it could be the IP of one of the machines, etc. Determine the hash values of these shards by passing their unique keys into the hashing functions.
- For each shard, there will be four hashed values corresponding to each hashing function.

- Consider the image below. It shows the shards (S1, S2, and S3) on the circle as per their hashed values.
- Let's assume **UserID** as the sharding key. Pass this UserID through a hashing function, **H,** which also generates output in the range [0, 10^18]. Let the output be **V**.
- Place this value, V in the same circle, and as per the condition, the user is assigned the first machine in the cyclic order which is **S3**.

Now, let's add a new shard **S4**. As per the outputs of hashing functions, let's place S4 in the circle as shown below.

- The addition of S4 shard helps us achieve a more uniform distribution of storage and load.
- S4 has taken up some users from each of the S1, S2 and S3 shards and hence the load on existing shards has gone down.



Note:

- Though the illustration has used a circle, it is actually a sorted array. In this sorted array, you will find the first number larger than the hashed value of **UserID** to identify the shard to be assigned.

- In the example above, UserID is used as the sharding key. In general, it can be replaced with any sharding key.

Manual Sharding

Let's consider/create a system **SulochanaDB** which has following properties:

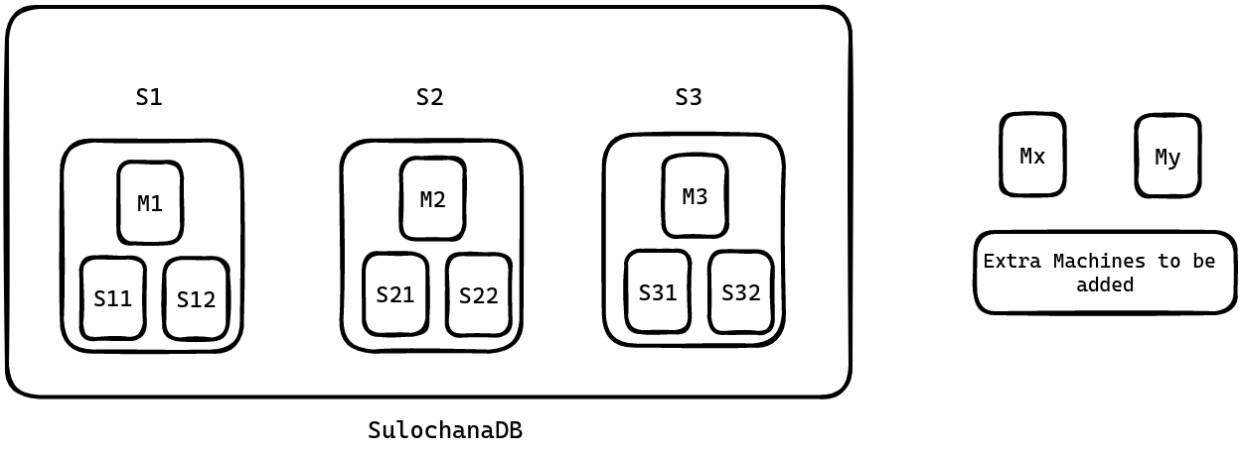
- It is initially entirely empty and consists of three shards S1, S2 and S3.
- Each shard consists of one master and two slaves as shown in the image below.
- Take any sharding key which will be used to **route** to the right shard. This routing is performed by **DB Clients** running Consistent Hashing code.
- Let's assume Sulochana is directed to shard S1. Now, she can use any of the three machines if only data read is required.
- However, the **write** behavior is dependent on what kind of system you want to create: **a highly-available** or **highly-consistent** system.

Open Questions

- How to implement the ability of adding or removing machines? Like how should the system change when new instances are added?
- What happens when a machine dies? What about the replication level?

Consider the following situation: You have two machines Mx and My which are to be added to **SulochanaDB**.

- What to do with these two machines?
  Options:
  - Add them to an existing shard.
  - Keep them in standby mode.
  - Create a new shard.



Manmeet's Algorithm

- Define an order of priority as follows:
    - Maintain the replication level (replace the crashed machines first). We have to first address the issue of under-replication. Reason behind this is we cannot afford the unavailability of the website. (Topmost priority)
    - Create a new shard.
    - Keep them in standby.
- Let's say we have **N** new machines and each shard consists of **M** machines.
    - Then N % M number of machines will be used for replacing crashed machines to maintain the replication level.
    - Remaining machines(divisible by M) will be used to create new shards.

Minor modifications discussed

- Let N = 3, M = 3 and currently one machine in S1 has died.
- But according to the algorithm, N % M = 0 machines are available to replace the dead machine.
- To solve this issue, we can decide a threshold number of machines, **X** which are always in standby to cater to our topmost priority, i.e. replacing dead machines and regaining replication level. This threshold can be a function of the existing number of shards.
- And from the remaining machines (**N - X**) we can create new shards if possible.

Note: Orchestrator implements these functionalities of maintaining reserve machines and creating new shards with the remaining ones. This Orchestrator goes by various names such as **NameNode** (Hadoop), **JobTracker**, **HBase Master**, etc.

Utilization of Standby Machines

- Contribution to existing shards by being slaves in them (additional replica).
    - If a slave dies in one shard containing one of these standby machines, you don't have to do anything as a backup is already there.

Now that we have got an idea of where to use additional machines, let's answer two questions:

- How are shards created?
- What is the exact potential number of reserve machines needed based on the number of shards?

Seamless Shard Creation

While adding a new shard, **cold start** (no data in the beginning) is the main problem. Typically, data migrations are done in two phases:

Staging Phase

- Nobody in the upper layer such as DB clients knows that there is going to be a new shard.
- Hence, the new shard does not show up in the Consistent Hashing circle and the system works as if the new shard does not exist at all.
- Now, a **Simulation** is executed to determine the **UserIDs** which will be directed to the new shard once it comes online.
- This basically determines the hash ranges that would get allocated to the new shard. Along with this, the shards which store those hashes are also determined.
- Now, let's say Staging phase starts at T1 = 10:00:00 PM and you start copying the allocated hash ranges. Assume at T2 = 10:15:00 PM the copying process is complete and the new shard is warmed up.
- However, notice it still may not have the writes which were performed between T1 and T2.
    - For example, if Manmeet had sent a write request at 10:01:00 PM then it would have gone for shard S1.
    - Let's assume Bx and By bookmarks were added by Manmeet at 10:01:00 PM. Now, there is no guarantee that these bookmarks have made their way to the new shard.

Real Phase

In this phase, the new shard is made live(with incomplete information).

- Hence you have to catch up with such relevant entries (made between T1 and T2). This catch up is really quick like a few seconds. Let's say at T3 = 10:15:15 PM, the catch up is complete.
- However, at T2 you made S4 live. Now, if Manmeet again asks for her bookmarks between **T2 and T3**, there are two choices:
    - **Being Highly Available:** Return whatever the new shard has, even if it is stale information.
    - **Being Highly Consistent:** For 15 seconds, the system would be unavailable. However, this is a very small duration (15 mins to 15 seconds downtime).

Timelines:
T1: Staging Phase starts.
T2: New shard went live.
T3: Delta updates complete. Missing information retrieved.

After T3, S4 sends signals to relevant shards to delete the hash ranges which are now served by itself (S4). This removes redundant data.

Note: Existing reserved shards are better tied to shards where it came from. Hence, these existing reserved machines could be utilized to create new shards. And the new machines can take up the reserve spot.

Estimate of the number of Reserved Machines

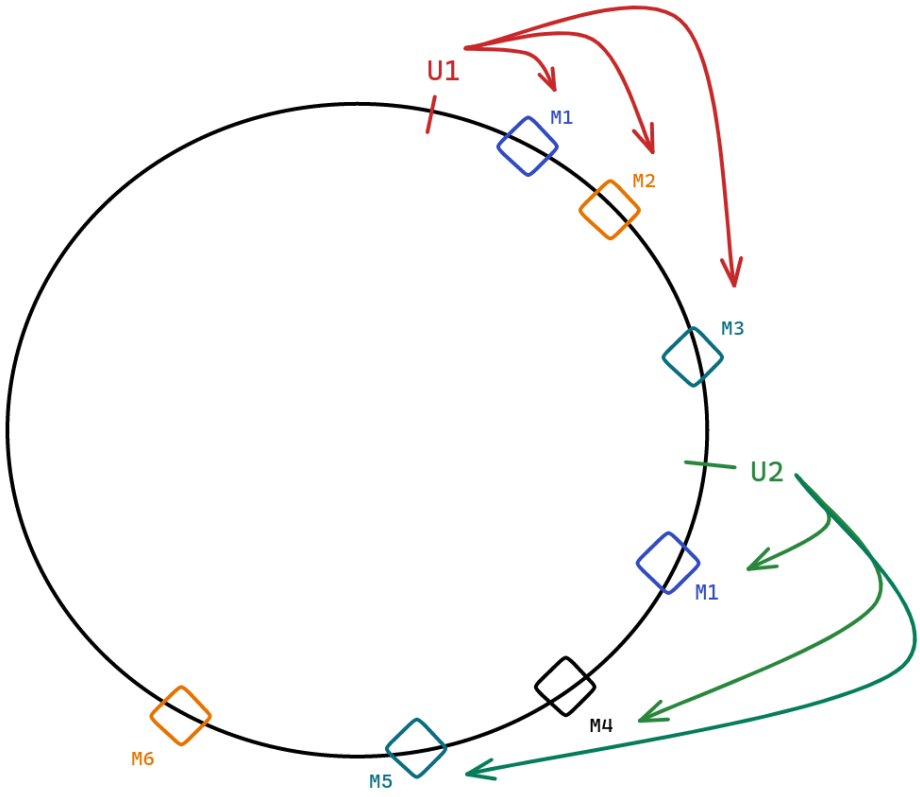
- Reserved Machines = X * Number of Shards
- Number of required reserved machines actually depends on the maximum number of dead machines at a time.
- Maximum number of dead machines at a time depends on various factors such as:
    - Quality of machines in use
    - Average age of machines
- Now, the approach to determine this number is to calculate
    - The probability of failing of X machines simultaneously
    - Expected number of machines dead at the same time
- There is another approach to this problem: Multi-master approach deployed by DynamoDB, Cassandra, etc.

Multi-Master

Consider a system of multi-master machines i.e. every machine in the system is a master. There is no slave. In the Multi-Master system, there is no need for reserved machines. Every single machine is a master and it is brought to the consistent hashing circle as and when it is live.

Master machines M1, M2, M3, etc. are shown in the Consistent Hashing circle.

- Now, let's say replication level = 3. Imagine a user U1 as shown below. Since you want to maintain three replicas, what are the optimal two machines where you should put bookmarks of U1?
- If M1 dies and U1 makes a request, which machine gets U1's request? M2 right. Hence, it would be better if M2 already had the second replica of U1's bookmarks.
- Finally, the third replica of U1 should be in M3 so that even when both M1 and M2 die, there is no struggle to find U1 data, it's already in M3.
- Remember, M2 should have a replica of only U1's bookmarks and not a complete replica of M1. Similarly for M3.
- To complete, U1's second replica should be in M2 and third replica should be in M3.
- Similarly, U2's second replica should be in M4 and third replica should be in M5.
- So, it makes sense that for a user, the three replicas should be in the next three **unique** machines in cyclic order.



Read, Write Operations

- In Multi-Master, you can have tunable consistency. You can configure two variables: **R** and **W**.
- **R** represents the minimum number of read operations required before a read request is successful.
- **W** represents the minimum number of write operations required before a write request is successful.
- Let **X** be the replication level. Then **R <= X** and **W <= X**.
- When R = 1 and W = 3, it is a highly consistent system.
    - Till all the machines are not updated, a write operation is not considered successful. Hence, highly consistent.
    - Even if one of these machines is taking time to respond or not available, writes will start failing.
- If R = 1 and W = 1, it is a highly available system.
    - If a read request arrives, you can read from any of the machines and if you get any information, the read operation is successful.
    - Similarly, if a write request arrives, if any of the machines are updated, the write operation is considered successful.
    - After the successful update of any one machine, other machines can catch up using the **Gossip** protocol.

    - This system may be inconsistent if a read request goes to a non-updated machine, you may get non-consistent information.
- In general,
    - As you increase R + W, Consistency increases.
    - **Lower R + W => Lower Consistency, Higher R + W => Higher Consistency**.
    - If **R + W > X**, you have a **highly consistent** system.
    - Because by just changing **R** and **W** it is possible to build a highly consistent or available system, it is also called **tunable consistency**.

The value of R and W depends on the type of application you are building. One of the frequent uses of DynamoDB is the Shopping Cart checkout system. Here:

- The shopping cart should be as available as possible.
- But if there should not be frequent cases of inconsistency and X = 5, then keeping R = 2, and W = 2 suffices. That way, you are writing to two different machines.
- If anytime you receive inconsistent responses from two machines, you have to merge the responses using the timestamps attached with them.

Example:
Response 1:
Lux Soap 10:00 PM
Oil: 10:15 PM

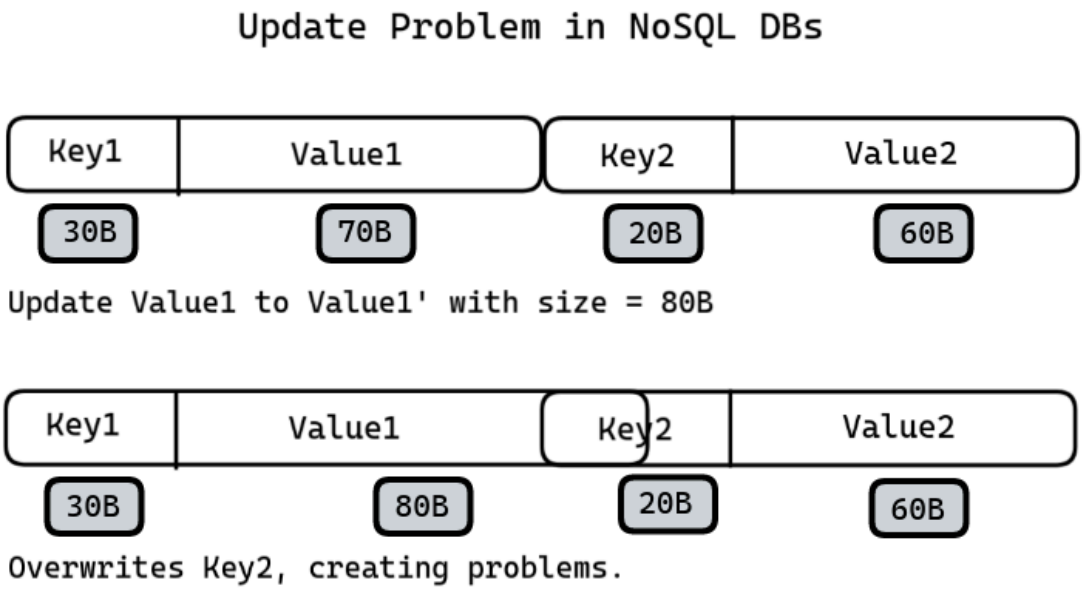
Response 2:
Lux Soap: 10:00 PM
Mask: 10: 20 PM

Merge:
Lux Soap
Oil
Mask

Questions for next class

- Storing data in SQL DBs is easy as we know the maximum size of a record.
- Problem with NoSQL DBs is that the size of value can become exceedingly large. There is no upper limit practically. It can grow as big as you want.
    - Value in Key-Value DB can grow as big as you want.
    - Attributes in Document DB can be as many in number as you want.
    - In Column Family, any single entry can be as large as you want.
- This poses a problem in how to store such data structure in the memory like in HDD, etc.

Update Problem



So, the question is how to design the data storage structure of NoSQL databases given the variable sizes of its records?

Points during discussion

- Sharding key is used to route to the right machine.
- One machine should not be part of more than one shard. This defies the purpose of consistent hashing and leads to complex, non-scalable systems.
- **Heartbeat** operations are very lightweight. They consume minimal memory (a few bytes) and a small number of CPU cycles. In the Unix machine, there are 65536 sockets by default and it uses one socket for its functioning. Hence, saving on heartbeat operations does not increase efficiency even by 0.1 percent.
- **CAP theorem** is applicable on all distributed systems. Whenever you have data across two machines and those two machines have to talk, then CAP is applicable.
- Rack aware system means the slaves are added from different racks. Similarly, Data Center aware system implies the slaves are added from different data centers.

Complexity of Write and Delete Operations

Delete operations are very fast as compared to write operations. You may have observed this when you transfer a file vs when you delete the same file. It is because:

- Delete operations do not overwrite all the bits involved. They simply remove the reference which protects the bits from getting overwritten.
    - That's why deleted files can be recovered.

- However, write (or overwrite) operation involves changing each of the bits involved, hence costlier.