Problem statement:

Designing a datastore which is efficiently scales after importing huge set of files as after importing many files the entire datastore will be too large to fit in memory. Records in the datastore should be unique by STB, TITLE and DATE. Subsequent imports with the same logical record should overwrite the earlier records.

Design Approach:

We can use **consistent hashing** as one of the ways to implement horizontal scaling and algorithmic cluster sharding.

Consistent Hashing is a distributed hashing scheme that operates independently of the number of servers or objects in a distributed hash table by assigning them a position on an abstract circle, or hash ring. This allows servers and objects to scale without affecting the overall system.

The logical placement of servers is done in circular manner so that each server stores the files as per the degree measures. The Filename (STB-TITLE-DATE) can be used as an input (Key) to the hashing function which gives output as a degree measure based on which file is stored and accessed from the designated server.

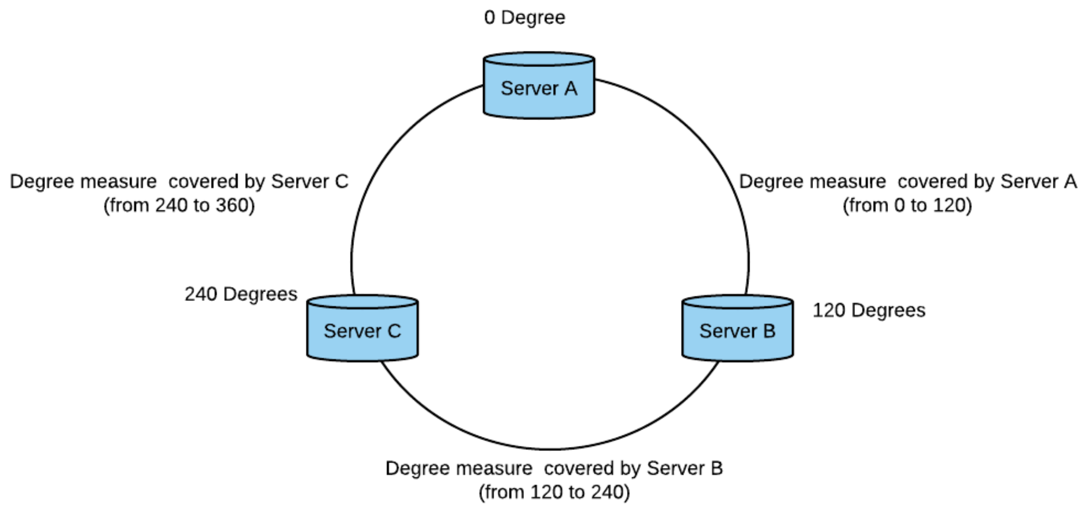
Assumptions:

1. The average load for importing the file into the datastore is around 20 Million per day
2. We have standard servers with storage capacity of 4TB per server

Initial scenario:

We have 3 servers initially which cover the below degree measure as per the diagram:

Server A - from 0 to 120 degrees
Server B - from 120 to 240 degrees
Served C- from 240 to 360 degrees

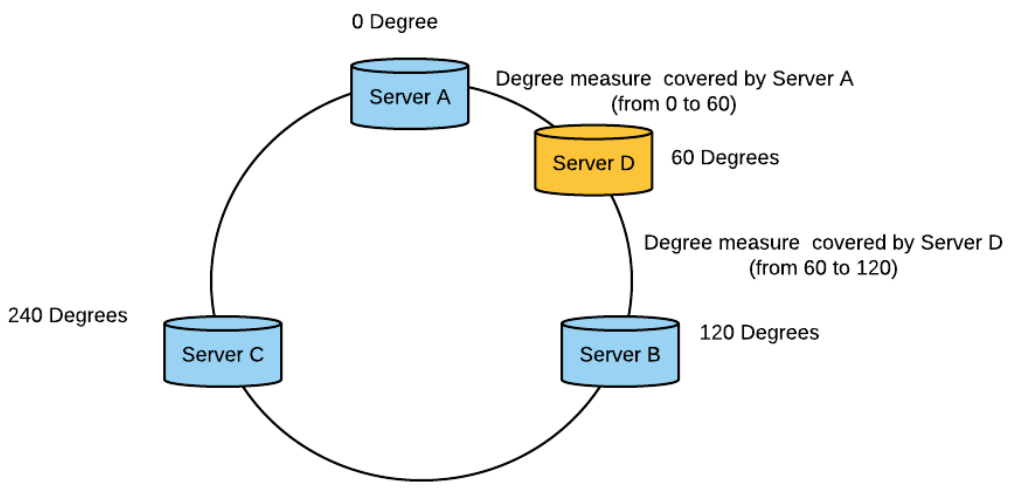

0 Degree
Server A
Degree measure covered by Server C (from 240 to 360)
Degree measure covered by Server A (from 0 to 120)
240 Degrees
Server C
Server B
120 Degrees
Degree measure covered by Server B (from 120 to 240)

Incoming files are stored on these servers based on the hashing function degree measure output. e.g - File with degree measure from 0 to 120 will be stored on Server A

Scenario in case of higher memory utilization:

Now let's say Server B & C are reached 50% of their capacity but Server A has reached 75% of its memory capacity. (Warning starts at 70%). We can put additional server D at 60 degrees So that the division of coverage becomes as below:

Server A - from 0 to 60 degrees
Server D- from 60 to 120 degrees
Server B - from 120 to 240 degrees
Served C- from 240 to 360 degrees

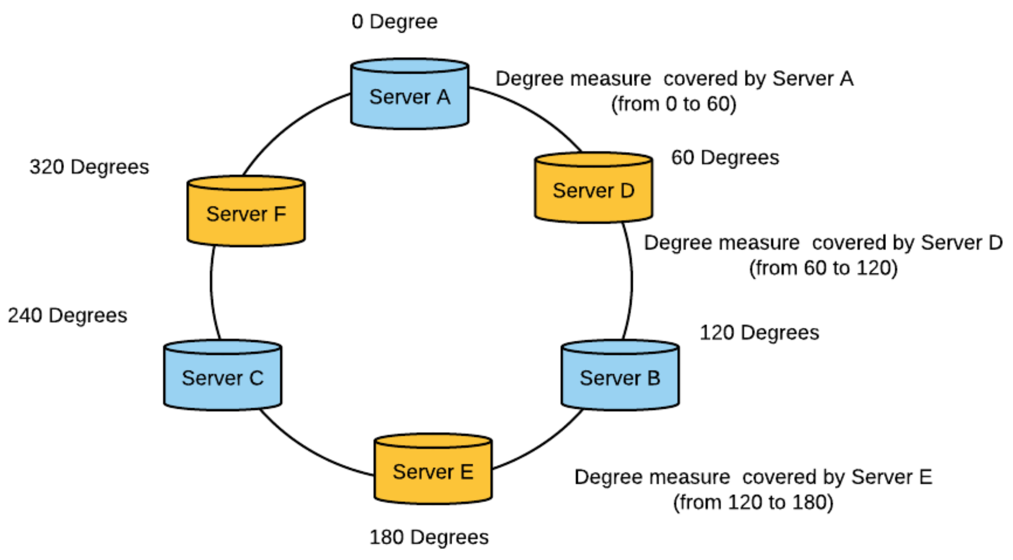

0 Degree
Server A
Degree measure covered by Server A (from 0 to 60)
Server D
60 Degrees
Degree measure covered by Server D (from 60 to 120)
240 Degrees
Server C
Server B
120 Degrees

<u>Storing a new file:</u>

In this case, let's say the hashing function outputs 90 degrees for a new file storage. Since it falls between 60 to 90, Server D will be used to store the file.

<u>Existing file is read/modified:</u>

When a file is requested to read or modify, the file is returned to the user, but the modified/read file is moved to Server D by rehashing the same. We can keep adding the general-purpose servers to this ring/circle instead of arranging for the expensive infrastructure as per below diagram


0 Degree
Server A
Degree measure covered by Server A (from 0 to 60)
60 Degrees
Server D
Degree measure covered by Server D (from 60 to 120)
120 Degrees
Server B
Degree measure covered by Server E (from 120 to 180)
Server E
180 Degrees
Server C
240 Degrees
Server F
320 Degrees

WHY consistent hashing:

1. We do not need to rehash everything in case of existing server failure or addition of new server. Re-hashing is only needed for existing files being accessed for read or modify operation.
2. Removing a server results in its object keys being randomly reassigned to the rest of the servers, leaving all other keys untouched:
3. Solving the rehashing problem. In general, only k/N keys need to be remapped when k is the number of keys and N is the number of servers
4. **Scalability with performance & Availability:**
   When using distributed caching to optimize performance, it may happen that the number of caching server changes (reasons for this may be a server crashing, or the need to add or remove a server to increase or decrease overall capacity). By using consistent hashing to distribute keys between the servers, we can rest assured that should that happen, the number of keys being rehashed—and therefore, the impact on origin servers—will be minimized, preventing potential downtime or performance issues.