Distributed Hash Tables

Advanced Algorithms (UE17CS311)

Aishwarya Kumaraswamy
PES1201701575

Madhav M Kashyap
PES1201700227

Priyanka Kumari
PES1201802433

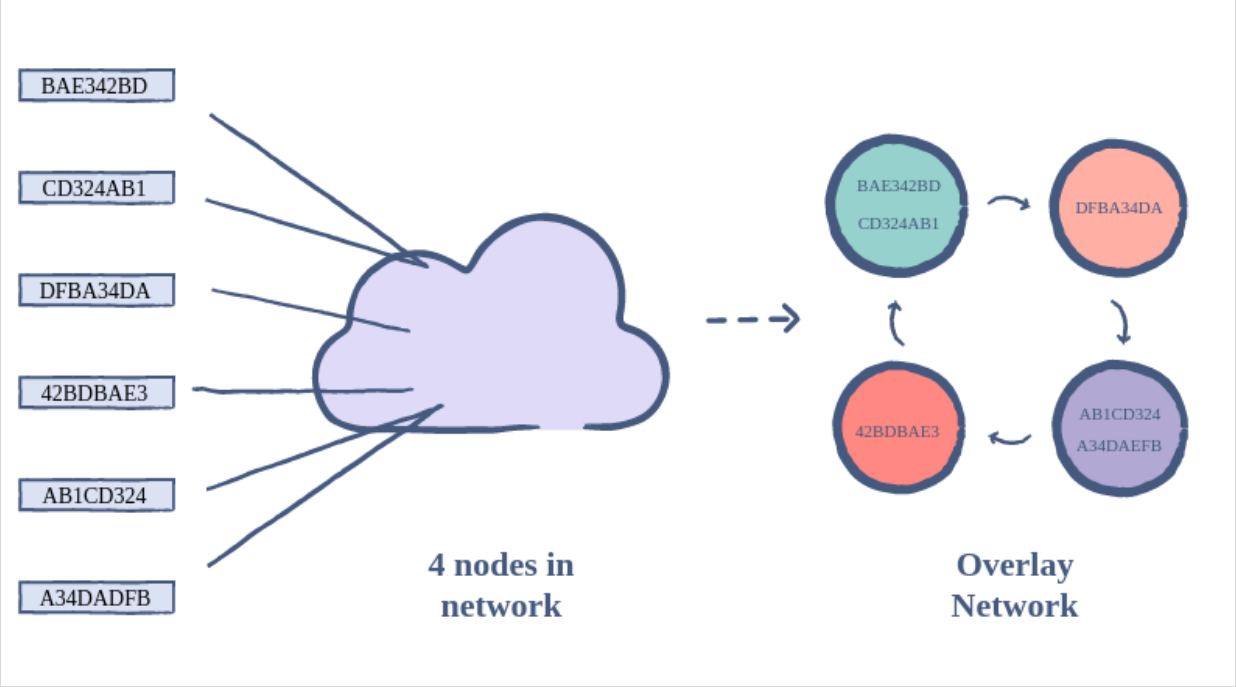
Distributed Hash Tables

In some situations, it may be necessary or desirable to split a hash table into several parts, hosted by different servers. One of the main motivations for this is to bypass the memory limitations of using a single computer, allowing for the construction of arbitrarily large hash tables (given enough servers). In such a scenario, the objects (and their keys) are distributed among several servers, hence the name, distributed hash tables.

> A distributed hash table (DHT) is a decentralized storage system that provides lookup and storage schemes similar to a hash table, storing key-value pairs. Each node in a DHT is responsible for keys along with the mapped values. Any node can efficiently retrieve the value associated with a given key. Just like in hash tables, values mapped against keys in a DHT can be any arbitrary form of data.

DHTs have the following properties:

- Decentralised & Autonomous: Nodes collectively form the system without any central authority.
- Fault-Tolerant: System is reliable with lots of nodes joining, leaving, and failing at all times.
- Scalable: System should function efficiently with even thousands or millions of nodes



Naive Hashing in Distributed Systems

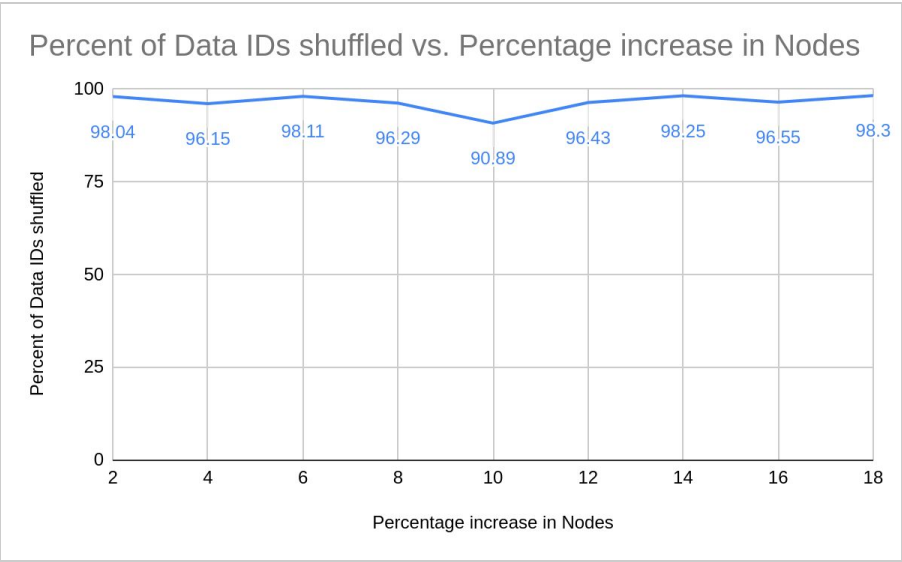
Now, how does distribution take place? What criteria are used to determine which keys to host in which servers?

The simplest way is to take the hash modulo of the number of servers. That is ***server = hash(key) mod N***, where N is the size of the pool. To store or retrieve a key, the client first computes the hash, applies a modulo N operation, and uses the resulting index to contact the appropriate server (probably by using a lookup table of IP addresses). Note that the hash function used for key distribution must be the same one across all clients, but it need not be the same one used internally by the caching servers.

Problems with Modulo Hashing Distributed Systems

The main problem with the naive approach of just taking modulo hashing is the Rehashing Problem:

- Let's suppose we change the number of nodes by adding/deleting nodes.
- Since the keys are distributed depending on the number of nodes, and the number of nodes has changed; the hash value of the new nodes are completely different from the previous node hash values.
- Thus, we see that the problem in a distributed system with simple rehashing is that a small change in the number of nodes will result in a huge amount of work to reshuffle all the data around the cluster. This becomes unsustainable as the amount of work required for each hash change grows linearly with cluster size.



As we can see, almost 90-98 % of the data IDs change their nodes during this process of increasing the number of Nodes. This shifting of data IDs results in a huge performance hit in large scale distributed applications.

Consistent Hashing

The above-stated problem can be solved by Consistent Hashing.

It uses a method called “Collision Hashing“ which calculates the Hash function of each node independently of the Number of Nodes in the ring.

This independent mapping prevents the need for repositioning the nodes every time a new node is added/deleted.

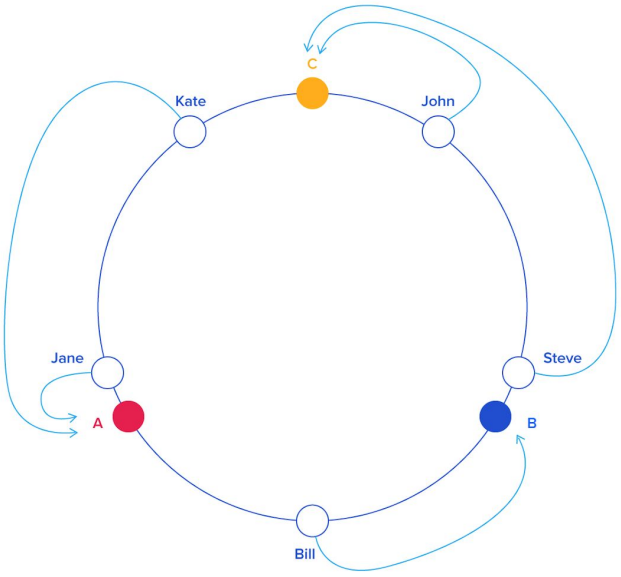
The Data IDs are mapped onto the next Node they find in Clockwise direction of the ring.

- The Hashing function I have implemented is ‘md5’ which translates an Integer ID into a real number between 0 to 1 inclusive.
- Position of server = Hash_Function (Server_ID) * Num_Data_IDS
- Data ID mapped to its next clockwise neighbouring node.

Whenever a Node is added, only the load balance of the 2 servers which the new server is added in between is affected. The new Node handles a part of the Data IDs the previous node in the Anti-Clockwise direction used to handle.

Similarly, when a node is deleted, only the server which lies next in the clockwise direction will be affected. The Data IDs which were handled by the Deleted node is pushed onto the clockwise next node.

The rest of the nodes do not change their positions and most of the data_IDs do not change their node mappings.



Problems with Naive Consistent Hashing

While the average theoretical load of each node still remains:

$$Average\ Load\ Balance = \frac{1}{N}$$

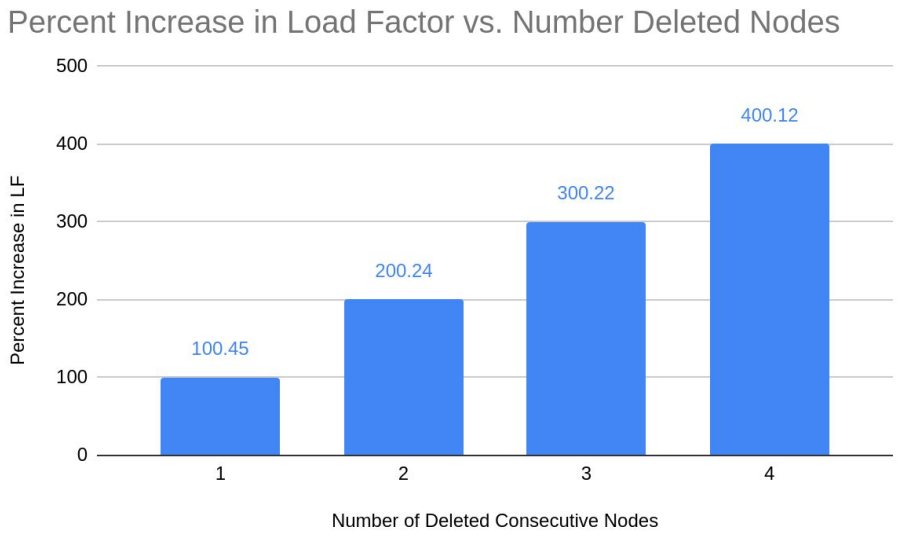
However, the load balances of individual nodes can be skewed very heavily above/below the expected average load balance.

For example: If a node is deleted, then the next node receives all of the Data IDs of the deleted node. This almost doubles the Load factor of the next node.

To demonstrate this, we plotted a graph of the Increase in Highest Load factor of the next node v/s the number of continuous nodes deleted.

Num_Nodes=100

Num_Data_Points=10000000

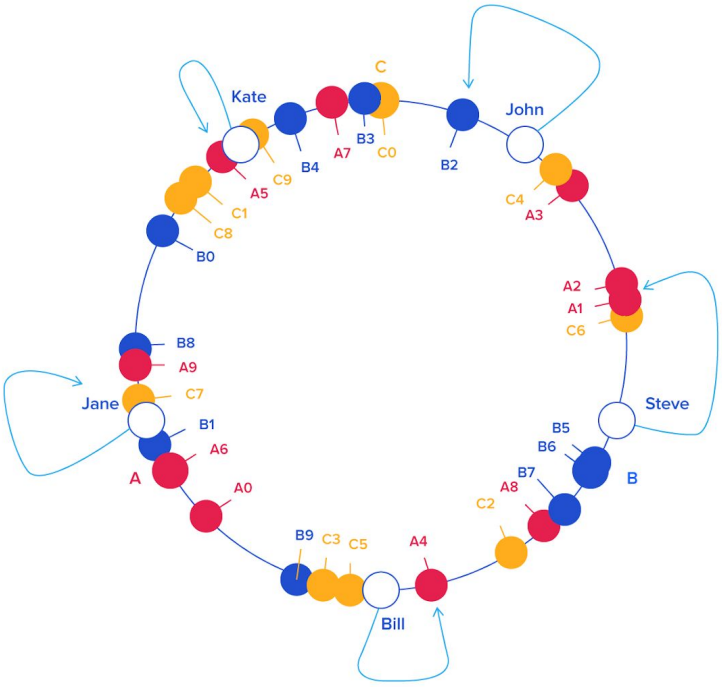


As we can see, removing continuous nodes in the ring makes transfers the load of the previous ‘n’ servers onto the next server clockwise. This makes the next server’s load increase disproportionately compared to the other nodes.

Consistent Hashing with Virtual Nodes

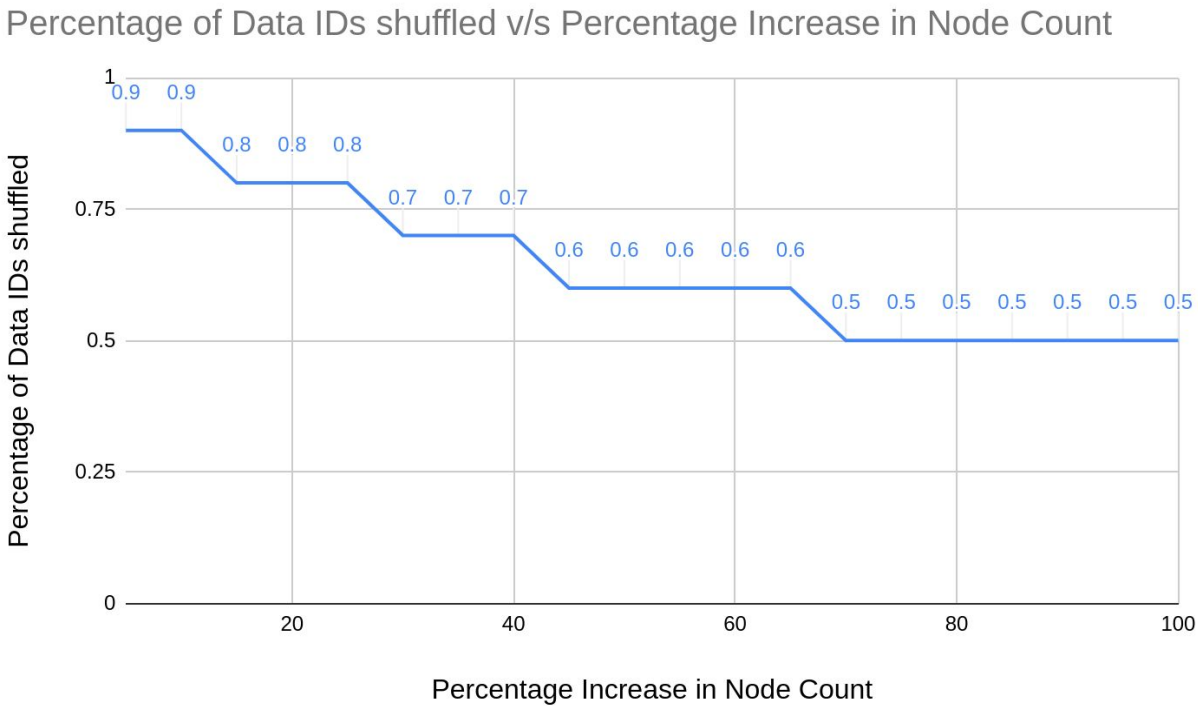
To ensure that one server is not heavily loaded if a node is deleted, we have to apply a simple trick:

- To assign multiple virtual servers to each individual server. So instead of having nodes A, B and C, we could have virtual nodes A0 .. A9, B0 .. B9 and C0 .. C9, all interspersed along the circle.
- The virtual nodes (aka replicas) A0, A1 .. A9 are all mapped to the same node A.
- If we delete A, we actually delete all the virtual nodes from A1 to A9.
- The extra load of deleting A is shared by 9 other virtual nodes.
- Therefore, no one node has a very high/low skew of load balance compared to the average load balance.



The net effect of load balancing among multiple virtual nodes is that a deletion/addition of a node has an effect on all the other nodes equally and to a lesser degree; compared with affecting only 1 node heavily in naive consistent hashing.

Num_Virtual_Nodes = 1000



The percentage of data IDS reshuffling using Virtual Nodes is only in the range of (0.5% to 0.9%) whereas, in the case of Distributed hashing, it was in the (90% to 98%) range.

Virtual nodes cause 100 times less reshuffling of Data IDs than Distributed Hashing!

This means that when adding/deleting nodes, Consistent hashing with Virtual nodes perform 100 times faster than Distributed Hashing.

The time complexity of search in consistent hashing with virtual nodes is of the order O(N)

Consistent Hashing with Binary Search Trees

Advantages of using a binary search tree instead of a hash table are:

- We can get all keys in sorted order by just doing Inorder Traversal of BST. This is not a natural operation in Hash Tables and requires extra efforts.
- Doing order statistics, finding closest lower and greater elements, doing range queries are easy to do with BSTs. Like sorting, these operations are not a natural operation with Hash Tables.
- With Self-Balancing BSTs, all operations are guaranteed to work in O(Logn) time. But with Hashing, Θ(1) is the average time and some particular operations may be costly, especially when table resizing happens.

Operations on a BST:

- To add a node: Find the hash of the node. If there is no node with that value, in the binary search tree, insert it at the right position.
- To delete a node: Find the hash of the node to be deleted in the binary search tree. If the node has no children, delete it. Else, find the in order successor of the node.
- To search for a key: Find the hash of the key. Return the next biggest element to the hash of the key in the tree.

Performance of BST compared to Hash tables: The time taken to search for the node in the BST is of the time complexity : O(log N), compared to the hash table approach where the keys are not sorted, which has O(N).

The below table shows the difference in average time taken in the two algorithms while searching.

Algorithm	Average time in seconds
Consistent Hashing with Hash Table	4.84784444173e-06
With BST	3.19480895996e-06