COMP 480/580 — Probabilistic Algorithms and Data Structure Jan 7, 2018

Lecture 7

Lecturer: Anshumali Shrivastava
Scribe By: Adjoa Asare (ena5), Joy Yu (jy92), Justin Park (jhp6), Michael Wong (mlw9)

1 Review of Consistent Hashing

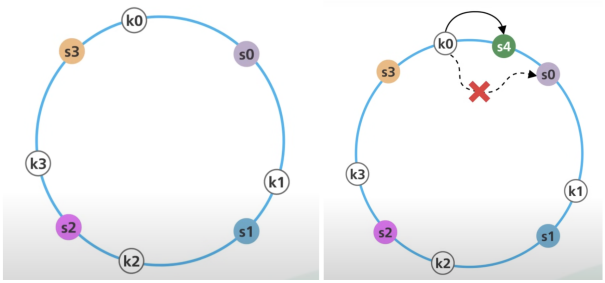
1.1 Description

Consistent hashing refers to a special type of hashing technique used to distribute data across multiple servers or nodes in a way that minimizes reorganization when nodes are added or removed. It is effective for situations where servers constantly come and go.
Goal: we want almost all objects to stay assigned to same server, even as number of servers changes

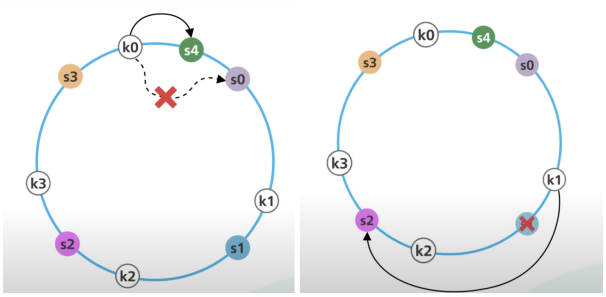
We achieve this by hashing the objects and server names using the same hashing function, resulting in a range of values from x_0 to x_n. This range creates a hash space that is visualized as a hash ring on which servers are placed. Each server is assigned a position on the ring based on its hash value. When assigning an object x, we compute $h_i(x)$ —the hash value of the object—and traverse the ring in a clockwise direction until we reach a server Y, which is defined by $h_m(Y)$. The object is then assigned to this server Y. In this configuration, each server is responsible for storing the objects located between its position and the position of its predecessor on the ring. This setup ensures that objects are assigned in a clockwise manner from their respective hash values.

1.2 Adding and Removing Servers

If we were to add a server s_4, only k_0 would need to be moved from s_0 to s_4 since s_4 is the first server k_0 encounters (going clockwise on the ring). With simple hashing, almost all of the keys would need to be remapped, whereas consistent hashing requires only a fraction of keys to be redistributed. (See below)



Similarly, if we were to delete s_1, only k_1 would need to be remapped from s_1 to s_2.



1.3 Optimizing with a Binary Search Tree

In consistent hashing, finding the correct server for a given object's hash value can be optimized using a binary search tree. Instead of scanning the hash ring linearly, we can use a binary search to quickly locate the closest server with a hash value that is greater than or equal to the object's hash value. For example, if the hash value of an object is 55, a binary search will efficiently find the next closest server's hash value in logarithmic time, O(log n). This ensures faster lookup times compared to a linear search, making consistent hashing scalable as the number of servers increases.

The hash ring should be significantly larger than the number of servers to ensure even distribution of objects and avoid clustering. To set up the ring, we can iterate through a range of values, hash them, and add these values as nodes to a binary search tree. When a request is received, the object is hashed, and a binary search is performed to find the server with the next highest hash value, ensuring the object is assigned to the correct server.

1.4 Benefits of Consistent Hashing

1. Minimal Data Movment: When a node is added or removed, consistent hashing ensures that only a small fraction of keys are relocated. This is because only the data items that would be mapped to the affected node are redistributed.

2. Scalability: Nodes can be dynamically added or removed without needing to rehash and redistribute all data. This property makes consistent hashing suitable for systems that frequently scale up or down, such as cloud-based applications.

3. Uniform Data Distribution: By using a good hash function, consistent hashing helps achieve uniform distribution of data across nodes, preventing hot spots or overloading of specific nodes.

1.5 Challenges

One major problem with consistent hashing is its vulnerability to cascading failures. For example, if multiple requests or loads hash to the same location, such as during a DDoS attack, the server at that location can become overloaded and crash. This failure can then "cascade" to other servers, causing them to become overwhelmed as they take on the extra load from the failed server. A common solution to mitigate this issue is to create dummy copies of the overloaded server (s_0), which helps distribute the load more evenly and reduces the risk of a cascading failure.

Consistent hashing is also challenged with data duplication when servers temporarily go offline or when new servers are added to the system. For example, when a new server joins, it

needs to be re-cached to store the data it is responsible for. Similarly, if a server is unused for a while, it may be deleted, as is typical in cache implementations.

2 SPOCCA

2.1 Semantics of Caching with SPOCA: A Stateless, Proportional, Optimally-Consistent Addressing Algorithm

The paper "Semantics of Caching with SPOCA: A Stateless, Proportional, Optimally-Consistent Addressing Algorithm" describes how Yahoo was able to change its video uploading system by changing how the front server caches. Before it was done by shared storages and load balances, which had their downsides such as cache misses and inefficient programs, however with SPOCA, it reduces the cache misses and also increases the efficiency of the program.

Before we get into the mathematical part of it, we would go into a real life example of how SPOCA works. Yahoo would want to upload their content into one of their servers and before we do that, Yahoo would assign their servers a proportion of the hash size. Then, the hash algorithm would hash the content name to the hash space. If the hash space is occupied by a server, it goes directly to the server. If not, it is rehashed until the server name is found. However, this can lead to problems such as cascading failure but we will describe the solution later in the notes.

To go into the mathematical aspect of it, let's say we have n objects (c_1, c_2, c_3 , ... , c_n). For the hash size, we should pick a size s that is at least greater than twice the size of all the objects added, or $s \geq 2 * \sum_{n=1}^{\infty} 2^{-n}$
We name the servers that are allocated s_1 through $s2$. Also, the range of the hash space consists of a number from 0 to the total hash size. To better illustrate this, let's say we have c_1 to c_n which is a size of 100. Then the total hash size would be at least 200. So if there is an object in the easy case if there exists a server in the space, we would allocate the object there. However, if there is not, we would need to hash the number that we hashed on the line. So for example if we hash into 160 and that does not exist, instead of hashing the object again, we would hash the value 160 and continue the process until we reach a hash value where a server exists.

This causes an issue where we have an infinite loop where a maps to b then b maps to a. The solution would be to include the number of retries in the hash function. To give an example, if we hash $h(o1, 0)$ and if a server does not exist, we do $h(h(o1, 0), 1)$ where the second parameter is the number of retries. So if $h(h(o1, 0), 1)$ does not exist in the hash space, we would perform $h(h(h(o1, 0), 1), 2)$. This would solve the issue of cascading failure because if a server is full, then the retry algorithm would hash to a different area and it ensures that they are generally mapped to different places because of the added retry values.

3 Data Streams

3.1 Overview

Data streams are a class of problems that require high speed. Thus, even 1-2 secs matter. Data processing needs to be faster than the data generation rate to keep up.

At this scale, there are certain problems:

1. Computational Efficiency
2. Memory

One examples of memory problems are keeping track of trending hashtags on Twitter. Hundreds of thousands of tweets are sent each minute. First, you have to filter out all hashtags and find the frequency of them. This requires a large amount of memory, which will quickly overwhelm the number of arrays for hash tables.

3.2 One Pass Model

One way to model data streams is to use the one pass model. In this model, you keep observing data streams and only have a small buffer, which is smaller than the data.

Essentially, data streams are represented by $U_1, U_2, \ldots, U_t$, where each U_i is some data at time i. There is a buffer B, which is some array of memory of size k, where k is less than the total amount of data. B should contain a random sample, meaning that the probability of any U_i being in B is $\frac{k}{t}$.

The one pass model is good for handling overwhelming traffic, as they are all data streams. Since there is so much information and we cannot store all of it, the one pass model uses a single pass on the stream. You first observe data, make a decision for the buffer, and then move on. However, anytime you need to look back and do a second pass, it won't work, as the stream has passed.

References

Ashish Chawla, Benjamin Reed, Karl Juhnke, Ghousuddin Syed. Semantics of Caching with SPOCA: A Stateless, Proportional, Optimally-Consistent Addressing Algorithm. USENIX-ATC'11: Proceedings of the 2011 USENIX conference on USENIX annual technical conference. 15 June 2011