Lecture Scribe

Eric Jia(ejj3), Alina Wang(jw146), Janet Jiang(jj65), Shreyan Das(sd100)

September 17, 2024

1 Context: two types of caches

1.1 Browser Cache

The browser cache is a client-side mechanism managed by web browsers to enhance the user experience by storing copies of frequently accessed web resources directly on the user's device. This caching process reduces load times for websites, as resources can be retrieved from local storage instead of being fetched from the server with each visit. By leveraging the browser cache, users benefit from faster response times and reduced bandwidth consumption, especially when revisiting sites. However, the cache is specific to the device and browser used; if a user accesses the same website from a different browser or device, the cache will not be available.

1.2 Web Cache

The web cache operates on the server side and is shared among multiple users, typically implemented within content delivery networks (CDNs) or proxy servers. Web caches store copies of frequently accessed web resources at strategic locations to minimize latency and reduce the load on the origin server. This shared approach allows users within a geographic region or network to benefit from faster access to cached content, leading to improved performance for large-scale websites and services. While web caches enhance efficiency and scalability for numerous users, they require more infrastructure and may introduce slight delays when cached content needs to be refreshed or updated.

1.3 Key Differences

Aspect	Browser Cache	Web Cache
Location	Client-side (on the user's device)	Server-side (proxy servers or CDNs)
Scope	Specific to individual users/devices	Shared across multiple users
Purpose	Improve browsing speed for a single user	Reduce latency and server load for many users
Infrastructure	Managed by the browser	Requires proxy servers or content delivery networks (CDNs)
Examples	Cached website assets in Chrome or Firefox	Akamai, Cloudflare, Netflix CDN

1.4 Bloom Filter Use: Web Caching

A more advanced use of caching involves the **Bloom Filter** in web caching, which is employed for optimizing frequently accessed web pages. For example, when you request a website like *amazon.com*, if you've accessed it recently, the page can be retrieved from a local cache. If not, a cache miss occurs, and the page must be downloaded from the server. This technique improves response times for the user while also reducing the load on servers and network traffic. Beyond individual benefits, web caching significantly reduces overall internet traffic, minimizing congestion, communication overhead, and packet loss.

1.5 Challenges of Web Cache that is shared by many users

Managing a shared cache for a large number of users presents several challenges. Remembering recently accessed web pages requires a considerable amount of fast storage for efficient retrieval, and Akamai's approach was to handle this in main memory. At scale, the cache needs to be distributed across multiple machines, raising the issue of deciding which machine to query for a specific resource. For instance, if there are 100 machines and a user requests amazon.com, it's inefficient to ask each machine individually, even with tools like Bloom filters. Thus, determining the optimal machine to poll first is key for minimizing latency and improving performance.

1.5.1 Intuitive attempt: simple hashing

In simple hashing, a resource like amazon.com is mapped to a specific server using a hash function (e.g., hash(amazon.com) = server ID), and the cache is retrieved from that server. This approach makes searching straightforward. However, it has significant drawbacks: when servers are added or removed, the hash function needs to be recalculated, leading to a complete reallocation of cached data. This dependency on knowing the exact number of servers becomes problematic, especially in dynamic environments where servers frequently change or fail. The limitations of this technique have been observed in large-scale data centers like Google's, highlighting the need for a more resilient solution.

2 Basics of Consistent Hashing

Imagine that we need to set up a data center. We are luckily able to procure a number of servers, with a need to store large amounts of data across these servers. At any point in time, a server might fail, and we cannot simply accept losing the data stored on that server. After all, this data is serving an important purpose. To make our clients happy, let us explore the method of consistent hashing.

2.1 Background

Consistent hashing is a hashing technique targeted towards distributed systems. In the 1990s, the idea of consistent hashing was first proposed but was ridiculed as just an idea that could not be put into practice. However, *Akamai* Technologies, named after the Hawaiian word meaning "clever,took a risk, incorporating consistent hashing into their web caches. Their triumph came in 1999 when the trailer for Star Wars: the Phantom Menacereleased exclusively on apple.com. The website crashed almost immediately, and those who desperately sought the movie trailer stumbled across Akamai's web cache. This demand made Akamai and consistent hashing much more relevant in the technology sphere. Since then, consistent hashing is used to address a variety of issues, such as those in P2P networks, and became a lightweight alternative to traditional databases.

2.2 Design

Much like the illustration with probing, consistent hashing puts servers and data in a ring of nodes. In other words, machines and objects belong to the same range,

occupying the same type of slots. Objects are hashed to a slot, using a hash function of the developer's choosing. Proceeding in one direction (let us say clockwise for the sake of the example), we will eventually come across a slot that is occupied by a machine. The object is then assigned to this machine. Note that this system is also resistant to server outages. All objects belong to the closest available machine in the clockwise direction, meaning there is built-in redundancy in the architecture. If most, if not all, of the machines are unavailable, there are more important issues to address.

2.3 Operations

The aforementioned method of insertion occurs linearly ($O(n)$) with respect to the size of the ring structure. This may lead to slower run-times than we hope for. Using a height-balanced binary search tree (BST) for the machines, we can make inserting and deleting operate in logarithmic time ($O(log(n))$). Let us investigate inserting and deleting objects and machines in turn.

- **Inserting object** x
    1. Hash x with function h_o to obtain $h_o(x)$
    2. Find the successor y of $h_o(x)$ in the BST
    3. Store x in y
- **Inserting machine** y
    1. Hash y with function h_m to obtain $h_m(y)$
    2. Find the successor y_s of y
    3. Move all objects x where $h_i(x) < h_m(y)$ to y
    4. Insert y into the BST
- **Deleting object** x
    1. Hash x with function h_o to obtain $h_o(x)$
    2. Find the successor y of $h_o(x)$ in the BST
    3. Delete x from y
- **Deleting machine** y
    1. Hash y with function h_m to obtain $h_m(y)$
    2. Find the successor y_s of y

3. Move all objects x where $h_i(x) < h_m(y)$ to y_s
4. Delete y from the BST

Considering hashing, insertion, and deletion to take place in constant time, notice that none of these operations do more than traversing the BST.

3 Load Balancing Math with Consistent Hashing

We have m items (or keys) and n machines (or servers). The goal is to distribute these items across machines using consistent hashing in a balanced manner, ensuring no machine is overloaded.

3.0.1 Expected Load per Machine

In a uniformly random hash space, each machine is expected to handle an equal portion of the items. The expected load per machine is simply the total number of items m divided by the total number of machines n:

$$\text{Expected Load per Machine} = \frac{m}{n}$$

3.0.2 Adding a New Machine

When a new machine is added to the system, the total number of machines becomes $n+1$. In a uniformly distributed hash space, the new machine will take over some of the items previously assigned to other machines. The expected number of items that will move to the newly added machine is:

$$\frac{m}{n+1}$$

This keeps the overall system balanced as machines are added.

3.1 High Probability Bounds on Load Distribution

3.1.1 Expected Fraction of Load for Each Machine

Ideally, each machine should hold a fraction $\frac{1}{n}$ of the total items in the system, where n is the number of machines.

3.1.2 Worst-Case Fraction of Load for a Machine

In the worst case, a machine could hold a larger-than-expected fraction of the load. Through probabilistic analysis, we can show that, with high probability, no machine will hold more than $O\left(\frac{\log n}{n}\right)$ fraction of the total items. This happens because the logarithmic term accounts for deviations from the expected uniform distribution and is generally small, so $\log n$ behaves almost like a constant.

3.1.3 Proof: Linear Probing Analogy

The proof follows the concept of *linear probing* in hash tables. When an item is hashed, if it falls into a hash interval that is already occupied (i.e., by a machine that already holds items), the algorithm moves to the next available interval (i.e., the next machine clockwise) until it finds an empty one.

- Assume the hash space is divided into n intervals corresponding to n machines.
- Each machine owns an interval l in the hash space.
- The remaining portion of the hash space, $1-l$, is divided among the other machines.
- The intervals are uniformly random, and the probability that any specific interval is empty depends on the number of machines.

3.1.4 Probability of Overloaded Intervals

Let's break the hash space into k sections of length l. The probability that none of the sections is empty (i.e., all sections are occupied by some machine) can be bounded as:

$$\Pr(\text{no section is empty}) = 1 - k(1-l)^n$$

Where $k(1-l)^n$ represents the probability that at least one section is empty. We want this probability to be small, ensuring that no machine holds too much load.

- This probability decreases exponentially as n grows, which shows that, with high probability, no machine holds more than $2l$ load.
- By setting $l = O\left(\frac{\log n}{n}\right)$, we ensure that no machine has more than this fraction of the total items, even in the worst case.

3.2 Birthday Paradox and Underloaded Machines

While we can say with high probability that no machine is overloaded, we cannot make the same claim about underloaded machines. Some machines may end up underloaded, but this situation is analogous to the *birthday paradox*. The paradox can be summarized like this:

- In a group of just 23 people, the probability that at least two of them share the same birthday is more than 50
- In a group of 57 people, the probability jumps to over 99

This counterintuitive result occurs because the problem isn't about one specific birthday being shared, but any birthday match between pairs of people. As the number of people grows, the number of possible pairs grows exponentially, leading to a higher likelihood of a shared birthday.

However, underloading is generally not a concern. An underloaded machine simply has unused capacity, which does not negatively affect the system's performance.

3.3 Conclusion

Using consistent hashing, we can ensure that, with high probability, no machine holds more than $O\left(\frac{\log n}{n}\right)$ fraction of the total items. This approach provides balanced load distribution and handles the addition or removal of machines efficiently, without having to redistribute all the data. Moreover, while some machines may be underloaded, this does not hinder the system and is analogous to the birthday paradox.