HLD Basics and Consistent Hashing

Contents

[Domain Name 3](#_Toc183622164)

[Domain Name Servers 5](#_Toc183622165)

[Dynamic and Static IP Address 6](#_Toc183622166)

[Load Balancer 9](#_Toc183622167)

[Consistent Hashing 14](#_Toc183622168)

**System design is not about knowledge; it's about interpreting scenarios, listing down corner cases, and building the right design for what we are creating!**

**An example of a system design problem:**

* The US hosts a large gaming event, the Super Bowl. During the Super Bowl ads, a website displays a QR code.
* The QR code provides a link to their website.
* On the website, tokens are assigned to users, allowing them to perform specific actions.
* **Question:**
  + When there is a large traffic load and you need to assign tokens to users,
  + How do you assign unique tokens to each user, and how does the website scale?
* We are going to take the example of a website called **del.icio.us**. This website existed back in 2004. The reason we are using this example is that the website went through a very interesting journey.
* The founder was a college student who started the website on a single laptop, just like many of us might begin a project.
* As the website started gaining popularity, he began to face numerous challenges, which led him to build a large-scale system to support it.
* Back in 2004–2005, the most popular internet browser was **Internet Explorer**. During those days, browsers did not have the concept of **user logins**. For instance, if you switched from one laptop to another, you wouldn't have the same browsing history or profile.
* Today, when you log into Google Chrome, you can access the same profile, settings, and history across devices after logging in. But back then, this was not the case.
* People relied heavily on **bookmarks** to navigate the web. The problem was that if you switched to a different device, all your bookmarks were lost.
* **del.icio.us** offered a very simple yet innovative feature. The idea was: you could log in to del.icio.us, and whatever bookmarks you saved would be stored in your account. Now, wherever you went—even to a friend's place—you could log into del.icio.us and access all your bookmarks. This ensured that you wouldn’t lose your bookmarks anymore.
* It was an extremely simple website with only two primary functions:
  1. Adding a new bookmark: The user could provide their user ID, the site URL they wanted to bookmark, and optionally a title to identify it.
  2. Retrieving bookmarks: The user could fetch all the bookmarks associated with their account.

AddBookmark(user\_id, site\_url);

GetAllBookmarks(user\_id);

* You could add more features later, such as updating or deleting bookmarks. Perhaps you might want to include a title for the bookmark or even update it with a different title.
* You might also want to delete bookmarks that you no longer find useful.
* For now, we are focusing on just two basic features: **adding** and **retrieving** bookmarks.
* So, we will build a website that stores bookmarks, and users can log in from anywhere to access them.
* It’s a very simple application. We will write the code on our laptop and implement two APIs: one for storing bookmarks and the other for retrieving them. These bookmarks can be stored in a local database.

#### The first problem

* Imagine there’s a random user on the internet. If they type **del.icio.us** in their browser and press enter, they should be able to connect to our laptop where our entire code is running.
* But how do these users know that they need to connect to this specific laptop whenever they type **del.icio.us**?

## Domain Name

**Step 1: Buying a Domain Name**

* The first step in setting up any website is to purchase a domain name.
* You can visit websites like **GoDaddy.com**, **Namecheap.com**, **domains.google.com**, or **BigRock.in** to check if a domain name is available.
* For instance, if we go to **GoDaddy.com**, we can search to see if the domain name **del.icio.us** is available.

**Questions:**

1. **How does GoDaddy.com know whether the domain name del.icio.us is available or not?**
2. **If the domain name is available, how do we purchase it? What happens next?**

**The Role of a Central Authority**

* All of this is possible because there is a central authority that maintains a record of all domain names in the world.
* This authority performs two key functions:
  1. It tells whether a domain name is available or already taken.
  2. If a domain name is available, it assigns that domain to a buyer when requested by an approved vendor (like GoDaddy).
* The central authority responsible for managing domain names is **ICANN** (Internet Corporation for Assigned Names and Numbers).
  1. ICANN is an American nonprofit organization that maintains the global database of domain names and their owners.
  2. It charges a fee for each domain name to cover its operational costs, such as maintaining its systems and infrastructure.
* Websites like **GoDaddy**, **Namecheap**, **domains.google.com**, and **BigRock** are resellers authorized by ICANN.

**Step 2: Mapping the Domain Name to an IP Address**

* Once we purchase the domain **del.icio.us**, it is not enough to simply own it. We also need a way to inform the world that typing **del.icio.us** should direct users to our machine.
* This is achieved by associating the domain name with an **IP address**.

**Why Use an IP Address?**

* In the machine world, devices identify each other using **IP addresses**. Every device connected to the internet has a unique IP address.
* You might wonder, "If everything is identified by IP addresses, why don’t we access websites using IP addresses directly?"
  + The reason is that IP addresses are difficult to remember, just like phone numbers.
  + To make it easier, we use domain names, which act as human-readable identifiers, while the system maps these names to corresponding IP addresses behind the scenes.

**Final Step: Configuring Ownership**

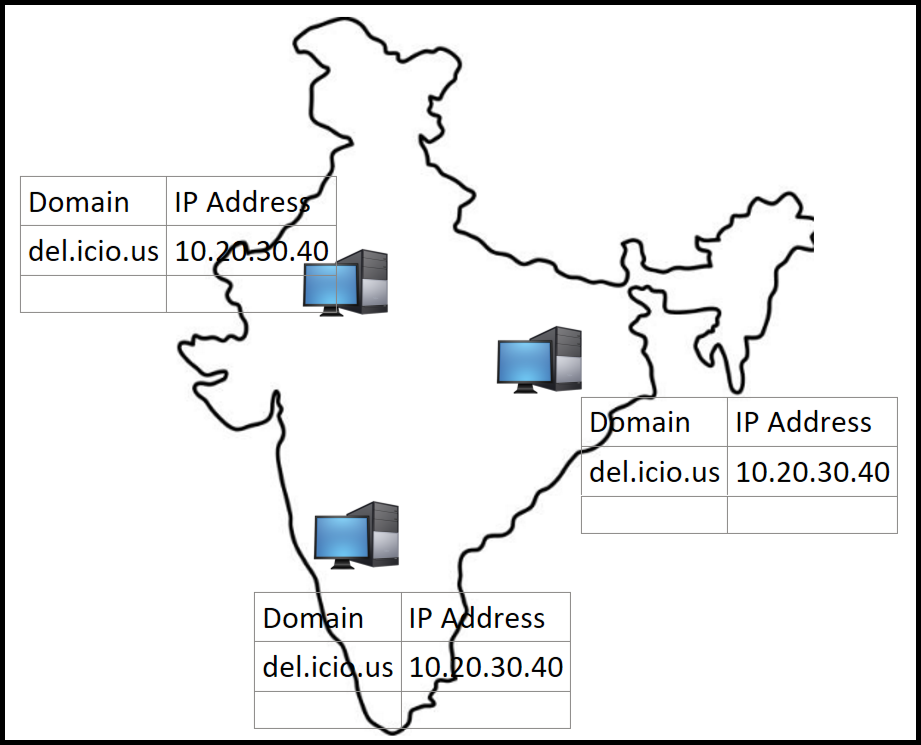
* As the owners of **del.icio.us**, we must configure the domain name to point to the correct IP address.
* This ensures that when someone types **del.icio.us** into their browser, the system routes them to our machine, where the website is hosted.

**GoDaddy.com** provides an interface where you can manage your domain settings. For example, it allows you to specify the IP address that people should connect to when they type your domain name.

* You could say, "Hey, set the IP address for **del.icio.us** to **10.20.30.40**."
* This information is then stored in the ICANN database, mapping the domain name to the specified IP address:

|  |  |  |
| --- | --- | --- |
| Domain | User | IP Address |
| del.icio.us | Nishith | 10.20.30.40 |

* Now, let’s say someone opens their browser and types **del.icio.us**. One way this could work is for every browser in the world to contact ICANN to find the corresponding IP address for **del.icio.us**.
* However, if ICANN were the only source storing IP addresses, consider the implications:
* Every browser on the internet would have to send a request to ICANN for every domain lookup (e.g., **Google.com**, **del.icio.us**, **Facebook.com**, etc.).
* Do you think this would be an efficient design? The answer is **no**—it would create a massive bottleneck and slow down the entire system.
* To improve performance and efficiency, another entity plays a key role: **Internet Service Providers (ISPs)**.
* ISPs, like **Airtel** in India, have machines (servers) distributed across their service areas.
* These servers maintain copies of the data stored in ICANN. This way, when you type a domain name into your browser, your request is typically handled by your ISP’s server instead of going directly to ICANN.



## Domain Name Servers

* These machines are called **Domain Name Servers (DNS)**.
* When we type a domain like **Google.com** or **Facebook.com**, the request goes to a DNS server to fetch the IP address of the website. These DNS servers are maintained by ISPs.

**How Do DNS Servers Get a Copy of ICANN’s Data?**

* If DNS servers fetched a complete copy of ICANN’s records every time, ICANN would quickly become overloaded. Millions of machines across the internet would constantly request massive amounts of data, which is impractical.
* To address this, ICANN uses a method called **differential fetch** or **incremental fetch**:
  + ICANN asks the DNS server, "When was the last time you updated?"
  + The DNS server provides the timestamp of its last update (e.g., **18 September, 9 PM**).
  + ICANN then sends only the entries that have changed since that timestamp.
  + This significantly reduces the amount of data transferred, as only a small subset of records is updated.
* After receiving the updates, the DNS server applies the changes and updates its own timestamp.

**Handling Newly Registered Domains**

* Imagine **del.icio.us** was just registered, and its IP address was added to ICANN. While one DNS server might have received the update, another DNS server (e.g., one managed by Airtel) might still be outdated.
* In such cases, if a user type **del.icio.us**, they might see an error message like "Site doesn't exist" because the DNS server hasn’t received the latest updates yet.

**Update Propagation Across the Internet**

* To manage this, a general expectation is set across the internet:
  + Any changes made to ICANN’s records—whether a new record is created, an existing record is updated, or a record is deleted—can take up to **six hours** to propagate across all DNS servers.
  + This delay ensures that updates are distributed systematically and efficiently without overloading the system.

**Note:** In most cases, DNS servers belong to your internet service provider (ISP).

* If you have never manually configured DNS settings on your device, you are likely using the DNS provided by your ISP by default.
* However, you can manually change the DNS settings on your laptop or device to point to a specific DNS server.
* There are other publicly available DNS servers maintained by organizations like **Google**, **Cloudflare**, and **Fast.com**, which you can choose to use if desired.
* If you haven’t changed your DNS settings, the default is typically the DNS provided by your ISP.
* My next step is to go to **GoDaddy.com**, reserve the domain, create a record for the IP address, and wait for 24 hours to ensure that all DNS servers across the world update with the IP address.
* Whose IP address do I provide? My laptop's IP address.

## Dynamic and Static IP Address

**How Does My Laptop Get an IP Address?**

* Typically, when any machine connects to the internet, the **internet service provider (ISP)** assigns a unique IP address to that machine.
* This IP address often changes each time you reconnect to the internet, which is why it’s called a **dynamic IP address**.

**The Problem with Dynamic IP Addresses**

* Imagine I connect my laptop to the internet. As the owner of **del.icio.us**, I check my IP address and see that it’s **10.20.30.40**.
* I go to **GoDaddy.com** and set the IP address for **del.icio.us** as **10.20.30.40**. Now, whenever someone types **del.icio.us**, they’ll be redirected to my laptop. Great! But there’s a problem:
  + If I disconnect and reconnect to the internet, my ISP might assign me a new IP address, for example, **10.20.30.50**.
  + What happens then? When someone types **del.icio.us**, their browser will attempt to access **10.20.30.40**, which is no longer my laptop’s IP address. This results in a **404 error** (site not found).

**Attempting to Fix the Problem**

* If I manually update the IP address in **GoDaddy.com** to the new one (**10.20.30.50**), it will still take several hours for the change to propagate to all DNS servers globally.
* During this time, users trying to access **del.icio.us** will face connectivity issues.

**Static IP Addresses**

* To solve this problem, I can opt for a **static IP address**.
  + A static IP address is a fixed address assigned exclusively to me by my ISP.
  + I pay an additional fee to my ISP (e.g., **Airtel**) for a leased connection with a static IP address.
  + With a static IP address (e.g., **10.20.30.40**), I am guaranteed that this IP address will not change, and it won’t be assigned to anyone else.

**What If My ISP Changes?**

* If I switch to a new ISP, I will need to purchase a new static IP address from them.

**Setting Up del.icio.us**

* As the owner of **del.icio.us**, I have now:
  1. Purchased the domain from **GoDaddy.com**.
  2. Set up the IP address entry with **GoDaddy**, ICANN, and DNS services.
  3. Secured a static IP address to ensure that my IP doesn’t change.

**How Does It Work Now?**

1. When someone types **del.icio.us** into their browser:
   * A computer screen shot

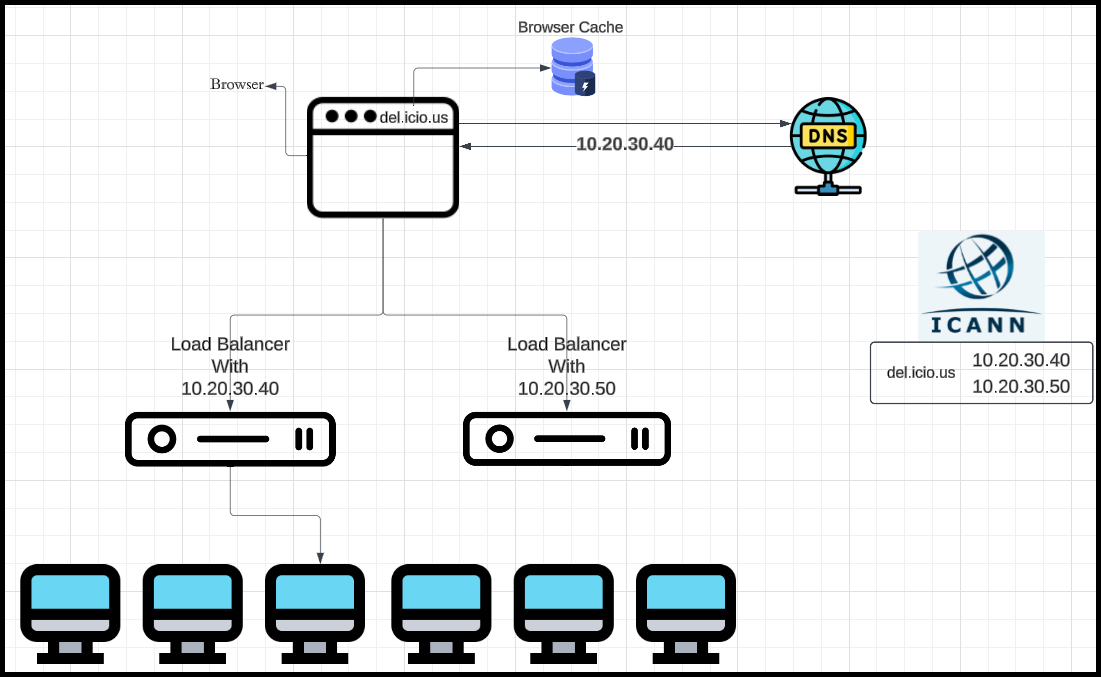
     Description automatically generatedThe browser first checks its cache to see if it has visited **del.icio.us** before and already knows the IP address. If it does, it uses that cached IP.
   * If not, the browser queries the DNS to find the IP address for **del.icio.us**. The DNS server responds with the static IP address (**10.20.30.40**).
2. The browser sends the request to **10.20.30.40**, which is my laptop’s IP address.
3. My laptop, running the necessary code, receives the request and serves the login page for **del.icio.us**.
4. After logging in, the website makes API calls to retrieve the user’s list of bookmarks. My laptop processes these requests, accessing the stored data to fetch and display the user’s bookmarks.

**Problems:**

1. **Code Deployment Causes Downtime**
   * If I want to modify my code—let's say I make some enhancements or optimizations—the code is already running on the machine. How do I update the running code?
   * First, I need to stop the currently running code. Then, I must copy the new code to the machine and start the updated version. During this process, from the time the old code is stopped to the time the new code starts running, there will be downtime. For example, if this process takes five seconds, my website, *del.icio.us*, will be inaccessible to users for those five seconds.
2. **Laptop Issues Lead to Downtime**
   * Since the website is hosted on a laptop, any issue with the laptop could cause downtime. For instance, if the laptop restarts due to a hardware or software problem, any user trying to access *del.icio.us* during that time will face an outage. The request will reach my laptop, but if the laptop is rebooting, it won’t be accessible.

**How would you solve this problem? How would you ensure there is no downtime?**

* By having a **Load Balancer.** A **Load Balancer** is a device or software that sits between clients (e.g., browsers) and backend servers. Its job is to distribute incoming requests across multiple servers or resources based on a set of predefined rules or algorithms.
* It acts as a single point of contact for clients but dynamically routes traffic to available servers in the backend.
* If one server (or load balancer in a multi-LB setup) goes down, the load balancer can seamlessly redirect traffic to the remaining operational resources.



* A user types **del.icio.us** in their browser.
* The browser queries the DNS to resolve the domain name into an IP address.
* The DNS responds with the IP address of the load balancer.
* The browser sends the request to the load balancer.
* The load balancer distributes the request to one of the healthy backend servers based on its algorithm.
* If one server goes down, the load balancer stops sending traffic to it and continues routing to the remaining servers.

## Load Balancer

* How Load Balancers Work?
* Let’s explore how load balancers operate. For simplicity, we’ll refer to a single load balancer for now, but keep in mind that there can be multiple load balancers in a system.
* What Should an Ideal Load Balancer Do?
  + The responsibilities of a good load balancer include:
  + Routing requests to the machines that are actively running the required code.
  + Distributing the load equally across all available machines.
  + Ensuring (on a best-effort basis) that all incoming requests are directed to live and functional machines.
* Additional Responsibilities of a Load Balancer
  + A load balancer may have additional responsibilities, especially in cases where the machines in the system vary in size and capability. For instance:
    - If the machines have different configurations, such as varying CPU power, storage capacity, or RAM, the load balancer should consider these differences.
    - For example, imagine a scenario where one machine has twice the CPU, storage, and RAM of another. The load balancer should allocate more traffic to the larger machine while assigning less to the smaller one to optimize resource utilization and ensure efficient processing.
* How to Achieve These Goals
  + To fulfill these responsibilities, a load balancer should:
    - Route requests to machines that are actively running the necessary code.
    - Distribute the load proportionally or equally based on machine capabilities.
    - Ensure (on a best-effort basis) that all incoming requests are directed only to live and functional machines.
* Load balancer should have (in memory) the list of all the machines and its status.

A diagram of a computer network

Description automatically generated

**Load Balancer Algorithms**

There are several standard algorithms for load balancing that you, as a user, can configure based on your needs. Examples include:

1. **Round Robin**
2. **Weighted Round Robin**
3. **Least Response Time First**

**Implementing Health Monitoring in a Load Balancer**

If we were writing the code for a load balancer, one critical task would be determining which machines are alive and which are dead. This check must be performed frequently to ensure proper load distribution. Two common approaches are:

**1. Health Check (Pull Method)**

* The load balancer periodically pings each machine to check its health.
* Example:
  + A machine exposes an endpoint like Del.icio.us/health\_check with a simple response, such as:

health\_check() { return 1; }

* + The load balancer calls this endpoint every few seconds.
  + If a response is received, the machine is marked as alive.
  + If no response is received, the machine is marked as dead.

**2. Heartbeat (Push Method)**

* Each machine is responsible for sending periodic "heartbeat" messages to the load balancer, indicating its health status.
* The load balancer monitors these messages:
  + Machines send "hello" messages every 5–10 seconds.
  + If the load balancer receives a message, the machine is marked as alive.
  + If no message is received within the defined time frame (e.g., 10 seconds), the machine is marked as dead.

**Pros and Cons of Each Method**

* **Health Check (Pull)**:
  + **Pros**: Simple to implement and centralized control.
  + **Cons**: Requires additional effort to set up health check endpoints and requests.
* **Heartbeat (Push)**:
  + **Pros**: Machines proactively communicate their status, reducing the load balancer's workload.
  + **Cons**: Requires every machine to implement and maintain the heartbeat logic.

Both methods have their use cases, and the choice depends on the specific system requirements and architecture.

If we revisit the **Del.icio.us** example, imagine all the laptops in the system were running some code while also storing the entire list of users and their bookmarks. This could amount to a significant volume of information.

Now, if each machine maintains a complete replica of this data, is that a good design? Let's consider the potential issues:

1. **Consistency Problems**
   * If a bookmark is updated on one laptop, and the next request goes to a different laptop, the user may not see the update due to inconsistent data across machines.
2. **Excessive Redundancy**
   * Storing identical copies of data on all machines leads to unnecessary duplication, wasting storage resources.
3. **Disk Space Constraints**
   * For example, if each bookmark uses 500 bytes of storage:
     + If Del.icio.us becomes highly popular and starts receiving **1 million new bookmarks daily**, it would require 500 MB of storage per day.
     + Back in the day, when 40 GB was the highest memory available on hard drives, such a growth rate would quickly fill up the available disk space.

**A Better Design**

To address these challenges, we can separate the storage of data from the machines running the code. This introduces a **database machine**, which centralizes data storage and resolves the issues of:

* **Data consistency**: All machines fetch data from a single, authoritative source.
* **Redundancy**: Eliminates unnecessary duplication of data.
* **Scalability**: The database machine can be optimized for storage and scaled as needed to handle increasing data volumes.

This design simplifies the architecture and improves reliability and efficiency.

A diagram of a server

Description automatically generated

* Currently, if all machines are storing the same data, the Application Server can communicate with any one of these storage machines. It can randomly choose any machine to fetch or store data.
* However, if all machines store the same data, the **system will eventually run out of storage space**.
* What if we could distribute the information across these storage machines? For example:
  + **100 GB** is stored on the first machine,
  + A different **100 GB** is stored on the second machine,
  + Another **100 GB** is stored on the third machine.
  + This way, the total available storage becomes **300 GB**, and if more storage is needed, we can simply add more machines to increase the capacity.
* **How can we achieve this?**
  + To distribute the data efficiently, we can split all the information into smaller, separate chunks and assign each chunk to a specific machine, ensuring that no single machine becomes overloaded.
* **How does this work?**
  + We can use **hashing** on the user\_id to determine which machine stores the data.
  + By splitting the data into chunks using a hashing function, some chunks go to the first machine, others to the second, and so on.
  + This ensures that all three machines have **different information**, and all bookmarks for a specific user are stored on a single machine (not spread across multiple machines).
* This process is called **sharding**, which involves splitting data into multiple parts. In this case, we are using **user\_id** as the key to shard the data.

**How Do I Shard Based on user\_id?**

One common approach is to use a modulo (%) operation. Here's how it works:

1. Take the user\_id and calculate its modulo with the number of machines.
   * For example, if there are 3 machines and the user\_id is 3, calculate 3 % 3 = 0.
   * This result determines the machine: in this case, the first machine.

However, this approach has limitations:

**What Happens When the Number of Machines Changes?**

* If the number of storage machines increases (e.g., from 3 to 4), the modulo calculation changes.
* As a result, requests using the same **user\_id** as the key may be redirected to a different machine, leading to inconsistent data access.

|  |  |  |
| --- | --- | --- |
| User\_id | User\_id%3 | User\_id%4 |
| 11 | 2 | 3 |
| 14 | 2 | 2 |
| 16 | 1 | 0 |
| 15 | 0 | 3 |

Here’s a corrected and refined version of the content for improved grammar, readability, and logical flow:

**How Do I Shard Based on user\_id?**

One common approach is to use a modulo (%) operation. Here's how it works:

1. Take the user\_id and calculate its modulo with the number of machines.
   * For example, if there are 3 machines and the user\_id is 3, calculate 3 % 3 = 0.
   * This result determines the machine: in this case, the first machine.

However, this approach has limitations:

**What Happens When the Number of Machines Changes?**

* If the number of storage machines increases (e.g., from 3 to 4), the modulo calculation changes.
* As a result, requests using the same user\_id as the key may be redirected to a different machine, leading to inconsistent data access.

**Alternatives to Modulo**

1. **Sharding Based on Alphabetical Order of user\_name:**
   * Data is assigned to machines based on the first letter of the user\_name.
   * For instance:
     + Users with names starting from 'A' to 'E' go to the first machine, 'F' to 'J' go to the second machine, and so on.

**Why This Isn’t a Good Approach:**

* + It can result in **load skew**, where some machines handle significantly more data or traffic than others.
  + For example, names starting with 'X' or 'Z' may be rare, so machines corresponding to those letters receive little to no traffic.

This imbalance, where some machines are overloaded while others are underutilized, is called the **Celebrity Problem**.

**Improving Sharding with a Load Balancer**

Another idea is to use a **load balancer** to maintain a mapping of which user is assigned to which machine.

**Challenges with This Approach:**

* If the number of users is very large, it may not be feasible to store the entire mapping in memory (e.g., in a hash map).
* If the mapping is stored on a hard disk, it introduces latency and slows down the system significantly.

**What Else Can We Do?**

* Instead of fixed approaches like modulo or alphabetical order, **consistent hashing** can be used. This method minimizes the number of keys that need to be reassigned when the number of machines changes, ensuring a more balanced and scalable system.
* Consistent hashing solves both the load skew and the machine scalability problems effectively.

## Consistent Hashing

* **Consistent hashing** is used to efficiently distribute data across multiple database machines (or shards).
* Machines (shards) are placed on a **circular number space**. This number space is very large, e.g., from 0 to 1018, and wraps around like a circle.

#### **How it Works**

1. **Hashing Machines (Shards):**
   * Each machine is assigned a **unique ID** (e.g., IP address, MAC address).
   * A **hash function** maps the machine's ID to a number in the circular space.
   * Example:
     + Machine M1 → Hash = 10000
     + Machine M2 → Hash = 1015
     + Machine M3 → Hash = 108

A diagram of a computer network

Description automatically generated

1. **Hashing User IDs:**
   * A user ID is passed through the same hash function, which produces a number within the same range.
   * The user ID's data is assigned to the **next machine in the circle** (cyclically).
     + User ID ‘1234’ → Hash = 12,000

A diagram of a computer network

Description automatically generated

1. **Storing and Retrieving Data:**
   * When a request comes for a user, the hash function determines the location of the user ID in the circle.
   * The data is fetched from the **first machine** in the clockwise direction from the hash value.

**Implementation**

* The circular space is implemented using a **sorted array** of machine hash values:
  + Example:

|  |  |  |
| --- | --- | --- |
| M1 | M2 | M3 |
| 10,000 | 108 | 1015 |

* To locate the appropriate machine:
  + Perform a search to find the **smallest hash value larger than the user hash** (First number in this array which is just bigger than the hash that I have just generated.).
  + If no larger value exists, assign the user to the **first machine** (wrap-around).

**Handling Changes in Machines**

1. **Adding a Machine:**
   * A new machine gets a hash and is inserted into the circle.
   * Only the data in the range between the new machine and its clockwise neighbour needs to be reallocated.
   * Example: Adding M4 with hash 1010:
     + Users previously assigned to M2 in this range are now assigned to M4.
2. **Removing a Machine:**
   * If a machine fails or is removed, its data is reallocated to its clockwise neighbour.

**Challenges**

1. **Load Skew:**
   * There is no guarantee that the hash distribution is uniform, which can lead to imbalanced loads on machines.
2. **Reallocation Costs:**
   * Adding or removing machines causes data reallocation, but consistent hashing minimizes this by only affecting a small portion of data.

**Example Usage**

* **Cassandra**, a distributed database, uses consistent hashing for data distribution across its nodes.

**Challenges in Basic Consistent Hashing**

1. **Load Redistribution:**
   * Adding a new machine only helps redistribute the load of one adjacent machine.
   * Removing a machine shifts all its load to one adjacent machine, potentially causing **cascading failures**:
     + Example: If m3 goes down, all its load moves to m2, potentially overloading m2 and causing it to fail, leading to a chain reaction affecting all machines.

**Improved Design: Using Multiple Hash Functions**

1. **Multiple Hashes for Each Machine:**
   * Instead of a single hash, each machine is assigned multiple hash values generated by different hash functions (H1, H2, H3, etc.).
   * These hash values spread each machine across multiple points on the hash circle.

A diagram of a computer network

Description automatically generated

**Placement on the Circle:**

* Example for machine M1:
  + H1(M1) = 10,000
  + H2(M1) = 106
  + H3(M1) = 1012
* Repeat this for all machines (M2, M3, etc.).

**Advantages of Multiple Hash Functions**

1. **Better Load Distribution:**
   * Machines are spread across multiple points, reducing the chances of load skew.
   * Users are assigned to the **nearest machine hash in the clockwise direction**, which evenly distributes requests.
2. **Handling Machine Failures:**
   * If a machine (e.g., M3) goes down:
     + The load from each of its hash points is redistributed among the nearest remaining machines.
     + Instead of all load moving to a single machine, it is shared across multiple machines (e.g., M1 and M2), mitigating cascading failures.
3. **Adding a Machine:**
   * When a new machine (e.g., M4) is added:
     + Its hash values are inserted into the circle, redistributing data from adjacent machines.
     + The load is pulled from multiple machines (M1, M2, and M3), ensuring a balanced redistribution.

**Key Takeaways**

* **Uniform Load Balancing:**
  + Using multiple hash functions for each machine creates a more uniform distribution of data and requests across all machines.
* **Improved Fault Tolerance:**
  + Machines failing or being added impacts only a subset of the data, avoiding catastrophic cascading failures.
* **Scalability:**
  + The system can scale effectively with minimal reallocation by distributing the load more evenly across existing and new machines.

**Data Migration**

* **When Adding a Machine:**
  + Move data associated with hash ranges that the new machine now owns.
* **When Removing a Machine:**
  + Reassign the data from the failing machine’s hash ranges to its clockwise neighbours.

**Timeline of Copying Data**

1. **At Timestamp T1 (21:00:00)**:
   * Started copying **U2** information from **DB1** to **DB2**.
   * Requests are still directed to **DB1**.
2. **At Timestamp T2 (21:03:00)**:
   * The copying process is completed.
   * Between **T2 - T1** (3 minutes), there might have been updates to **U2** that are not copied to **DB2** yet.
   * If a request is sent to **DB2**, the information may be incomplete (some bookmarks might not be found).

A screen shot of a computer

Description automatically generated

**Types of Systems:**

**1. Highly Consistent Systems:**

* **Behaviour**:  
  If a new request arrives during the copying process (**T2 - T1**), the system will return a failure response until the copying is complete.
* **Advantage**:  
  Ensures consistent and accurate data.
* **Example**:
  + Banking Systems
  + Messaging Systems

**2. Highly Available Systems:**

* **Behaviour**:  
  Requests during the copying process will not fail, as the system prioritizes availability. However, the data might be incomplete (e.g., missing bookmarks).
* **Advantage**:  
  Ensures the system remains operational even during partial data updates.
* **Example**:
  + Most Social Media Systems.