HLD Caching - 1

Contents

[Summary of HLD Basics and Consistent Hashing 2](#_Toc183724615)

[Machine with Code and Storage 3](#_Toc183724616)

[Current System Status 5](#_Toc183724617)

[Rule of Optimization 6](#_Toc183724618)

[Caching 7](#_Toc183724619)

[Optimizing the First Load 9](#_Toc183724620)

[Image Upload and Delivery Process in Instagram (Using CDNs) 11](#_Toc183724621)

[Cache Eviction and Eviction Algorithms 14](#_Toc183724622)

[Cache Invalidation and Synchronization 18](#_Toc183724623)

## Summary of HLD Basics and Consistent Hashing

The question is: **Why is system design even needed?**

* Why can't things be built on a single machine?
* Why do you need so many components?
* We started with the example of a website called **Delicious**, a bookmarking website from 2004, where people could store their bookmarks and access them from anywhere.
* We began by discussing how the website started in a hostel room on a single laptop. Then, we explored why there was a need to move to multiple machines.
* This led us to understand the need for a **load balancer**.
* We also explored the exact setup process:
* What is **DNS**?
* What is a **load balancer**?
* What happens when you start running out of space?
* What do you do next?

In that context, we discussed **sharding** as a solution.

And within the context of sharding, we also covered the concept of **consistent hashing**.

**Notes:**

1. **Why System Design is Important:**
   * **Single Machine Limitations**: Building systems on a single machine becomes inefficient as traffic and data grow.
   * **Need for Multiple Components**: As the system scales, there’s a need for different components to handle various tasks (e.g., load balancing, data storage, etc.).
2. **Example of Delicious (2004):**
   * Delicious was a bookmarking website that allowed users to store and access their bookmarks from anywhere.
   * Initially, it was hosted on a single laptop in a hostel room, but scaling it to handle more users and data required moving to multiple machines.
3. **Need for Multiple Machines:**
   * As the system grows, the data and traffic increase. A single machine can't handle the load, and hence, multiple machines are required.
   * This leads to the implementation of a **load balancer** to distribute the load across machines.
4. **DNS and Load Balancer Setup:**
   * **DNS** (Domain Name System): It translates user-friendly domain names to IP addresses, directing traffic to the correct server.
   * **Load Balancer**: It distributes incoming traffic across multiple servers to ensure no single machine is overwhelmed.
5. **Handling Scaling Issues:**
   * As data grows, machines start running out of space. In such cases, a load balancer helps redirect traffic, but additional solutions like **sharding** are needed to efficiently handle the growing data.
6. **Sharding:**
   * **Sharding** is the process of splitting data into smaller chunks (or shards) that are distributed across different machines. This helps scale the system without overwhelming any single machine.
7. **Consistent Hashing in Sharding:**
   * **Consistent Hashing** is a method used in sharding to ensure that data is evenly distributed across machines, even when the number of machines changes.
   * It helps prevent issues such as uneven data distribution and minimizes rebalancing when machines are added or removed.

## Machine with Code and Storage

A computer and a server

Description automatically generated with medium confidence

**Issues with Coupling Code and Storage on the Same Machine:**

1. **CPU vs. Memory-Intensive Processes:**
   * **Code Execution** is CPU-intensive, meaning it uses the CPU heavily when running.
   * **Storage Access** is memory-intensive, requiring both memory and some CPU resources when looking up and returning data.
2. **Impact of Increased Traffic:**
   * If a website experiences a surge in traffic (e.g., many users fetching their bookmarks), there’s a need to scale.
   * If both the **code** and **storage** are on the same machine, multiple machines will be required, each with its **own local code and storage running**.
   * This leads to **data duplication** across machines, causing several issues:
     + **Higher Cost:** Storing the same data on multiple machines is costly, as you’re maintaining multiple copies of the data.
     + **Consistency Issues:** Ensuring that all the machines are in sync can be challenging, leading to potential data inconsistencies.
3. **Managing Data Consistency:**
   * Storing the same data on many machines may not be necessary; perhaps 2-3 copies would suffice to ensure reliability.
   * Having 100 copies of the same data on different machines increases storage costs and complicates the synchronization process.
4. **Application Code Deployments:**
   * Developers often want to **improve code** by adding new features or making fixes. Frequent deployments may be required.
   * If the code and storage are on the same machine, every deployment requires restarting both the application and the storage, which is inefficient.
     + **Problem:** Even though the storage hasn’t changed, it must be restarted, resulting in wasted resources as the storage isn't utilized during the deployment process.
5. **Service Restarts and Downtime:**
   * If you tightly couple storage and application code on the same machine, restarting the service for code updates means **downtime** for the storage as well, even though it wasn’t impacted by the update.
   * This causes **wastage** of resources, as storage sits idle while the application code is redeployed.
6. **Deployment Failures:**
   * If bad code is deployed (e.g., causing exceptions), it could lead to failures. Machines running the faulty code would stop, and both the application and storage would be unavailable on those machines, causing additional downtime.

**Advantages of Coupling Code and Storage:**

* The **primary advantage** of keeping code and storage on the same machine is **low latency**. Fetching data from RAM (local storage) is faster than fetching it from disk or across the network.
  + **Improved Latency:** There are fewer network calls, and data can be accessed directly from the local disk or RAM, reducing latency.

## Current System Status

A diagram of a server

Description automatically generated

**Separation of Concerns**:

* Systems separate the **application layer** and **storage layer** for flexibility, better optimization, and data consistency.
* This is why applications, storage, and servers typically run on separate machines.

**Deployments Without Downtime**:

* Use a **load balancer (LB)** to manage requests among application servers.
* During deployment:
  + Notify the LB to remove the server undergoing deployment.
  + Update code and restart the server.
  + Re-register the server with the LB after it's ready.
* Requests are handled by other servers while one is being updated, preventing downtime.

**Health Checks**:

* LBs periodically check server health. Without manual intervention, delays in health checks can lead to brief failures during unplanned restarts.

**Real-World Deployment Practice**:

* Deployment happens in batches, one machine at a time.
* CICD or manual processes ensure smooth transitions using configurations and pre-defined steps.

## Rule of Optimization

* **Core Principle**: Optimize the **slowest step first**, as it yields the highest returns.
* Focus on identifying the component or operation that takes the most time and prioritize optimizing that.

**Scenario: Accessing a website (e.g., Delicious)**

1. **User Request**:
   * User types **del.icio.us** in the browser and presses enter.
   * The browser begins loading an HTML page, which might contain:
     + Text (e.g., "Hello, Rahul").
     + Images.
     + CSS and JavaScript files (to make the site dynamic).
     + User-specific content like bookmarks (if logged in).
     + Login page (if not logged in).

A computer screen shot showing a diagram

Description automatically generated

1. **System Architecture**:
   * The user is located in **India**, using a DNS server provided by their ISP (e.g., Airtel). Assume this DNS server is in **Mumbai**.
   * The del.icio.us infrastructure:
     + A **Load Balancer** in the **United States** (e.g., near New York).
     + Application servers and storage systems hosted in the same **US Data Centre** (all close to each other for fast internal communication).
2. **Communication Flow**:
   * The request travels across multiple components:
     + **Browser → DNS**: The user’s browser queries the DNS server to resolve the domain name into an IP address.
     + **DNS → Load Balancer**: The DNS directs the browser to the load balancer's IP address.
     + **Load Balancer → Application Servers**: The load balancer forwards the request to one of the app servers.
     + **Application Servers → Storage**: The app servers query the storage layer for required data.

**Which two components take the most time to communicate?**

* The answer depends on **physical distance**:
  + The **browser (India)** and **load balancer (US)** are the **farthest apart**.
  + This is because the transcontinental transfer of data introduces the highest latency due to long physical distances and potential network bottlenecks.

**What can I do to make that faster?**

## Caching

**Baseline Problem**:

* Transferring even a small amount of data from the browser to the load balancer could take **500ms** due to the distance.

**Solution: Caching**

* **Concept**: Store a **local copy** of frequently accessed data closer to the user to avoid repeated fetching from a distant source.
* Analogy:
  + Fetching tea leaves from a distant tea estate every time you want tea is inefficient.
  + Instead, you keep a stock of tea at home for quick access.

**Caching in Practice**

1. **DNS Caching Example**:
   * When the browser resolves del.icio.us for the first time:
     + It queries the DNS server for the IP address (e.g., 10.20.30.40).
   * For subsequent requests (within a certain time frame), the browser stores this IP mapping in a **local cache**.
2. **Performance Gains**:
   * Resolving through DNS might take **10ms**, but using a cached local copy takes **1ms**.
   * This results in a **90%-time savings** for each subsequent request during the cache validity period.
3. **Key Concepts in Caching**:
   * **Cache Invalidation**:
     + Ensuring the cached data remains valid over time.
     + Introduces a mechanism to decide when cached data should expire.
   * **Time to Live (TTL)**:
     + A parameter defining how long cached data remains valid.
     + For example, caching the DNS resolution for 10 minutes prevents unnecessary re-fetching during this period.

**Why First Load is Slow but Subsequent Loads are Fast**

1. **First Load**:
   * When you visit a website (e.g., Instagram) for the first time:
     + Your browser has to fetch images, JavaScript, and other resources from remote servers.
     + These resources are loaded for the first time and stored in the browser's **cache**.
   * Resources like images are fetched by their **URLs**, and the browser saves a local copy.
2. **Subsequent Loads**:
   * When you refresh the page, the browser checks its cache:
     + If the resource's URL matches the cached version, it uses the local copy instead of fetching it again.
   * This significantly reduces loading time for resources already fetched.

**Browser Caching**

* **How it works**:
  + The browser stores copies of fetched resources locally.
  + Example: An image with a specific URL (example.com/image1.jpg) is saved.
  + On the next visit, the browser checks if the URL is cached and serves the cached resource.
* **Managing Cache Size**:
  + The browser ensures the cache doesn't grow indefinitely by using:
    - **Eviction policies**: Old or less-used resources are removed when space is needed.
    - **Time to Live (TTL)**: Specifies how long a resource should be cached before checking for updates.

**Challenges with the First Load**

* Even with browser caching, the **first request** needs to fetch all resources from the origin servers, which can be geographically distant (e.g., New York to India). This leads to higher latency.

## Optimizing the First Load

**Content Delivery Network (CDN):**

1. **What is a CDN?**
   * A **Content Delivery Network** consists of servers distributed globally that store copies of website resources.
   * Examples: **Akamai**, **Cloudflare**, **CloudFront**, **Fastly**.
2. **How it works**:
   * Companies like Facebook or Instagram upload their resources (e.g., images, videos, scripts) to a CDN.
   * The CDN stores these resources in multiple **data centres** around the world.
   * When a user requests a resource:
     + Instead of fetching it from the origin server (e.g., in New York), the CDN serves it from the closest data centre to the user (e.g., Mumbai or Singapore).
3. **Process Flow**:
   * Facebook uploads an image (image1.jpg) to the CDN.
   * The CDN assigns a unique URL for the image (e.g., cdn.akamai.net/image1.jpg).
   * Facebook's HTML page references this CDN URL.
   * When you load the page, your browser fetches the image from the **nearest CDN server**.
4. **Key Benefits**:
   * **Reduced Latency**: Resources are served from nearby locations.
   * **Faster First Load**: Heavy files like images and videos load quickly.
   * **Global Scalability**: CDNs handle high traffic by distributing requests across their network.

**How CDNs Work Behind the Scenes**

1. **Global Server Network**:
   * CDNs like Akamai and Cloudflare maintain servers in strategic locations worldwide (e.g., Mumbai, Singapore, London).
   * These servers store cached copies of resources.
2. **Caching Logic**:
   * Resources are cached at CDN servers only when requested.
   * If a CDN server doesn’t have the requested resource, it fetches it from the origin server, caches it, and serves it to the user.
3. **Identifiers**:
   * When a file is uploaded to the CDN, it is assigned an identifier or URL (e.g., cdn.akamai.net/file123.jpg).
   * Websites use these CDN URLs in their HTML, enabling browsers to fetch directly from the CDN.
4. **Dynamic Updates**:
   * When the original resource changes, the CDN automatically updates its cache using **cache invalidation** or **versioning mechanisms** (e.g., adding timestamps to URLs).

**User Experience Flow**

1. **First Load**:
   * The browser fetches the website's HTML from the origin server.
   * The HTML contains links to resources hosted on the CDN.
   * Resources are loaded from the nearest CDN server, reducing latency. When a user requests a resource, the CDN serves it from the nearest server using **Anycast** networking.
     + The closest server acknowledges the request and delivers the resource.
2. **Image Loading Behaviour**:
   * Initially, placeholders (empty boxes) are displayed while images load.
   * If the internet connection is fast, images load quickly, filling these boxes.
3. **Subsequent Loads**:
   * The browser uses cached versions of resources for faster rendering.

## Image Upload and Delivery Process in Instagram (Using CDNs)

**1. Image Upload Process**

1. **User Action**:
   * User uploads an image to Instagram.
2. **Facebook Load Balancer**:
   * The upload request is routed to a Facebook load balancer.
   * The load balancer forwards the request to one of the application servers.
3. **Application Server Processing**:
   * Recognizes the request as an image upload.
   * Encrypts the image for security (using public-private key encryption).
   * Stores the encrypted image in **object storage** systems like **Facebook’s Haystack** or **AWS S3**.

A computer screen shot showing a diagram

Description automatically generated with medium confidence

1. **Object Storage**:
   * Designed for large file storage.
   * Stores the image and provides a file path to retrieve it later.
   * Example: If you provide a file path, the object storage retrieves the file without interpreting its contents.
2. **Database Entry**:
   * The server creates a new entry in the **Post Table** in the database:
     + **Post ID**: Unique identifier for the post.
     + **Image Path**: File path in the object storage.
     + **CDN URL**: URL provided by Akamai (e.g., static.akamai.net/path/to/image).
3. **Upload to CDN**:
   * The server uploads the image to **Akamai CDN**.
   * Akamai assigns a **global CDN URL** and ensures the image is distributed across its network.
   * Multiple resolutions of the image may be uploaded for optimization (e.g., thumbnails, medium, full-size).

**2. Image Delivery Process**

1. **User 2 Requests Instagram Feed**:
   * User 2 (a friend of User 1) requests their Instagram feed.
   * Facebook retrieves the relevant posts, including User 1's post.
2. **HTML Response**:
   * Facebook sends back an **HTML page** that:
     + Includes lightweight text and layout.
     + Contains the **CDN URL** for the image instead of the image itself.
3. **Browser Fetches Images**:
   * The browser sends a request to Akamai to fetch the image using the provided CDN URL.
   * A computer screen shot of a computer

     Description automatically generatedThe request is routed using **Anycast** to the nearest Akamai server.

**3. How Akamai and Anycast Work**

* **Akamai’s Role**:
  + Stores and distributes images to its global servers.
  + Ensures fast delivery by keeping a copy of the image close to users geographically.
* **Anycast Networking**:
  + When a browser requests an image, it broadcasts the request.
  + The nearest Akamai server acknowledges and serves the request.
* **Single CDN URL**:
  + There is **one global URL** for the image (e.g., static.akamai.net/someimage.jpg).
  + The browser doesn’t know the server location—it reaches the nearest server using Anycast.

**4. Performance Optimization**

1. **Caching**:
   * Akamai caches images closer to users for faster access.
   * First-time access may be slower for a region, but subsequent requests are faster as Akamai caches the image locally.
2. **Load Distribution**:
   * Akamai’s global servers balance traffic to prevent overloading.
   * This ensures consistent performance, even under heavy load.
3. **Multiple Resolutions**:
   * By storing different resolutions, Akamai delivers the most appropriate image based on the device and network speed.

**5. Key Advantages**

* **Faster Image Load Times**:
  + Users retrieve images from the closest Akamai server.
  + Reduces latency compared to fetching directly from Facebook’s data centers.
* **Scalability**:
  + Akamai handles global traffic, allowing Facebook to focus on application-level concerns.
* **Resilience**:
  + Even if Facebook’s servers experience downtime, images remain accessible via Akamai.

Caching is an essential technique to enhance system performance by storing frequently accessed data closer to the requesting layer. It can be implemented at various levels: browser, CDN, application server, and even internally within machines (e.g., in memory). Below are the key concepts and problem statements for caching:

**Caching Levels:**

1. **Browser-Level Caching:**
   * Stores static assets (like CSS, JS, and images) on the user's browser to reduce load times for subsequent requests.
2. **CDN Caching:**
   * Distributes static content (e.g., images, videos) to servers geographically closer to users.
   * Uses techniques like **anycast** for efficient routing to the nearest CDN server.
3. **Application Server Caching:**
   * Frequently accessed data (e.g., popular user bookmarks) is cached in memory at the application server level.
   * Reduces the need to query slower storage systems repeatedly.
4. **Internal Machine Caching:**
   * Storage machines also implement caching by keeping frequently requested data in RAM instead of accessing the hard disk.
   * Increases retrieval speed within the storage system.

**Caching Problem Statements:**

1. **Cache Size Limitation:**
   * Caches are inherently small due to the use of faster but expensive storage like RAM.
   * **Key Question:** What happens when the cache is full?
2. **Cache and Storage Consistency:**
   * Cached data is a copy of the source-of-truth data in the storage system. The source data can change, making the cache stale.
   * **Key Question:** How do we keep the cache consistent with the storage system or invalidate stale cache entries?

## Cache Eviction and Eviction Algorithms

When a cache reaches its capacity, it must decide which entry to remove to make space for new data. This process is called **cache eviction**, and the strategy used to decide which entry to remove is defined by an **eviction algorithm**. Below are key points and details for notes:

**What is Cache Eviction?**

* **Definition:** The process of removing an entry from the cache when it is full to make room for new entries.
* **Reason for Eviction:** Caches are limited in size and cannot store all data.

**Eviction Strategies:**

There are two main approaches when the cache is full:

1. **Reject the new entry:** Do not add the new entry and return an error or fallback to the source.
2. **Evict an existing entry:** Remove one or more entries to accommodate the new entry.

**Popular Cache Eviction Algorithms:**

1. **LRU (Least Recently Used):**
   * **Definition:** Removes the entry that has not been accessed for the longest time.
   * **Rationale:** Entries that are not recently accessed are less likely to be used again soon.
   * **Use Case:** Suitable for scenarios where actively used data should remain in the cache (e.g., user bookmarks).
   * **Implementation:**
     + Use a combination of a **doubly linked list** and a **hash map** to track usage and maintain order.
     + Overhead: Requires extra memory and computational cost to update timestamps for each access.
2. **FIFO (First In, First Out):**
   * **Definition:** Removes the oldest entry in the cache (i.e., the one added first).
   * **Rationale:** Old entries have had sufficient time in the cache and are likely less critical now.
   * **Use Case:** Suitable for simple applications or when recency of access is not critical.
   * **Implementation:**
     + Can be implemented using a **queue**, which makes it very easy and has no additional memory overhead.
3. **LFU (Least Frequently Used):**
   * **Definition:** Removes the entry that has been accessed the least number of times.
   * **Rationale:** Entries that are rarely accessed are less likely to be needed again.
   * **Use Case:** Suitable for scenarios where the frequency of access is more important than recency.
   * **Implementation:** Requires maintaining access counters for each entry.
4. **Random Eviction:**
   * **Definition:** Removes a random entry from the cache.
   * **Rationale:** Simple and works well in scenarios where access patterns are unpredictable.
   * **Use Case:** Low-priority data or when simplicity is preferred.
5. **MRU (Most Recently Used):**
   * **Definition:** Removes the most recently accessed entry.
   * **Rationale:** Suitable for scenarios where recent access patterns suggest the least likelihood of quick re-access.
   * **Use Case:** Specialized cases like systems with sequential access patterns.

**Algorithm Selection:**

* The choice of eviction algorithm depends on the **access patterns** and **requirements** of the application.
* **Considerations:**
  + Memory overhead: Algorithms like LRU require more memory than FIFO.
  + Computational overhead: More complex algorithms (e.g., LFU) may slow down the cache.
  + Application type: For example, **LRU** is ideal for active user sessions, while **FIFO** is better for simple, queue-like data access.

**Summary:**

* **Eviction is necessary** when a cache is full and must free space for new entries.
* **Eviction algorithms** like LRU, FIFO, LFU, MRU, and Random Eviction help determine which entry to remove.
* The best algorithm depends on the application's data access patterns and performance requirements.

### Determining the Ideal Cache Size

The ideal size of a cache depends on several factors, including the system's memory, application requirements, and the number of active users. Below are key points to consider:

**Factors Influencing Cache Size**

1. **Available System Memory:**
   * The amount of free memory after accounting for application and system processes determines how much can be allocated for caching.
   * Example:
     + **Total RAM:** 8 GB
     + **Application Usage:** 6 GB
     + **System Processes:** 1–1.5 GB
     + **Available for Cache:** ~500 MB
2. **Entry Size and Capacity:**
   * The size of each cached entry dictates the total number of entries the cache can hold.
   * Example:
     + Cache Size: 500 MB
     + Entry Size: 1 KB
     + Capacity: **500,000 entries**
3. **Type of Caching System:**
   * If using an in-memory cache like **Redis**, the cache size will be constrained by the storage or memory available on the machine running Redis.
   * Example: If Redis can store X entries, eviction must occur when this limit is reached.

**Application Considerations for Cache Sizing**

1. **Active Users at Peak Times:**
   * The cache should be large enough to support all **active users** at a given time.
   * For example, if the application has 100,000 active users at peak and each user requires 10 KB of cached data, the cache must accommodate 1 GB.
2. **Access Patterns:**
   * Analyse typical user activity and the frequency of data requests to determine what portion of data needs caching.
3. **Scalability:**
   * Ensure the cache size can scale with user growth or handle peak loads without excessive evictions.

**General Guidelines:**

* **Memory Allocation:** Leave enough memory for system and application processes to prevent performance degradation.
* **Eviction Mechanism:** Choose an appropriate eviction algorithm (e.g., LRU, FIFO) to efficiently manage cache capacity.
* **Load Testing:** Use simulated loads to evaluate if the current cache size meets real-world performance requirements.
* **Periodic Monitoring:** Continuously monitor cache hit rates and adjust size or policies if required.

## Cache Invalidation and Synchronization

**Part 1: Cache Invalidation**

1. **Cache Invalidation Purpose:**
   * Ensures that the cache reflects the most recent and accurate data.
   * Avoids inconsistencies between the cache and the database.
2. **When Cache Invalidation May Not Be Critical:**
   * If data doesn't change frequently, temporary inconsistencies can be tolerated.
   * Example: **DNS Caching** (IP address rarely changes).
3. **Using TTL (Time-To-Live):**
   * **Definition:** Time period after which cached data is removed.
   * Helps maintain consistency over time by expiring outdated data.
   * Example: DNS caching with a TTL of 30 minutes:
     + Cache stores domain-to-IP mapping for 30 minutes.
     + After 30 minutes, the entry is invalidated, forcing a DNS lookup.
4. **Handling Failures in Cached Entries:**
   * If a cached IP fails, the entry is removed from the cache immediately.
   * The next access will fetch updated data (e.g., from the DNS).

**Part 2: Synchronizing Cache and Database (Server Side)**

1. **Challenges:**
   * Cache and database can go out of sync if updates are not coordinated.
   * Synchronization ensures that the cache reflects the latest database state.
2. **Caching Paradigms for Synchronization:**
   * **Write-Through Cache:**
     + Ensures cache and database are always in sync.
     + **Steps:**
       1. Update the cache.
       2. Update the database.
       3. Roll back the cache update if the database write fails.
     + **Pros:** Strong consistency between cache and database.
     + **Cons:** Slower writes due to dual updates.
   * **Write-Back Cache:**
     + Optimized for faster writes by deferring database updates.
     + **Steps:**
       1. Update the cache.
       2. Acknowledge the write request as successful.
       3. Update the database asynchronously.
     + **Pros:** Fast write operations.
     + **Cons:** Risk of data loss if the cache crashes before database update.
     + Suitable for analytics systems where speed is prioritized over strict consistency.
   * **Write-Around Cache:**
     + Writes are performed directly on the database, bypassing the cache.
     + Cache is updated only during reads (if data is not present in the cache).
     + **Pros:** Simplifies writes and avoids cache consistency issues.
     + **Cons:** Increased read latency if the cache is frequently outdated.
     + **Use Case:** Works well with TTL for cache consistency.

**Key Takeaways:**

1. **Cache Invalidation Methods:**
   * Use TTL for data that doesn’t change often.
   * Remove entries immediately on detection of failures.
2. **Caching Paradigms:**
   * **Write-Through:** Strong consistency but slower writes.
   * **Write-Back:** Fast writes but less consistent and risks data loss.
   * **Write-Around:** Simplified writes, potentially inconsistent cache.
3. **Use Cases:**
   * Use **write-through** for critical data requiring strict consistency.
   * Use **write-back** for analytics or real-time updates where speed matters more.
   * Use **write-around** for systems with less frequent reads or data updates.