HLD - Caching - 2

Contents

[Problem: Code Submissions on Scaler 2](#_Toc184027376)

[CDN vs. Local File Storage: 4](#_Toc184027377)

[Caching the Rank List 12](#_Toc184027378)

[Cache Hit and Cache Miss: 14](#_Toc184027379)

[Facebook’s Newsfeed on Profile Pages 16](#_Toc184027380)

[Back-of-the-Envelope Estimation for Newsfeed Caching 21](#_Toc184027381)

[Caching Strategy for Newsfeed 22](#_Toc184027382)

[Managing Load and Refresh for Newsfeed 24](#_Toc184027383)

## Problem: Code Submissions on Scaler

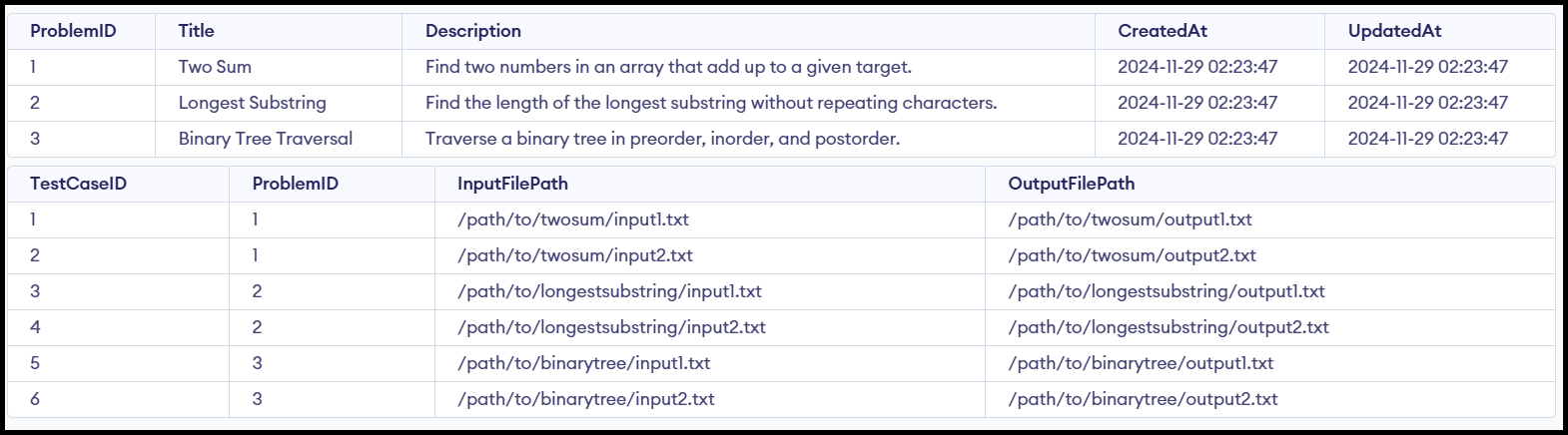
Consider the case of submitting DSA problems on Scaler. When you submit a problem on Scaler, the browser communicates with Scaler's load balancer, and the submission is routed to one of the many app servers. The app server receives the user ID, problem ID, code, and programming language.

To execute the code, the server requires the input file and the expected output file for the given problem. These files can be large, and fetching them from file storage takes time (approximately 2 seconds). This delay makes code submissions slow.

How can you make the process faster?

**Code Evaluation Flow at Scaler**

1. **Overview**:
   * A user submits code through the browser.
   * The browser communicates with Scaler's **Load Balancer**.
   * The **Load Balancer** forwards the request to one of the available **App Servers**.
2. **Components of a Submission**:
   * The submission contains:
     + **User ID**: Identifies the user submitting the code.
     + **Problem ID**: Identifies the problem being solved.
     + **Code**: The submitted solution.
     + **Programming Language**: Specifies the language of the submitted code.
3. **What the App Server Does**:
   * The **App Server** processes the request and decides if it can directly execute the code.
   * To evaluate the code, it requires:
     + **Input File**: Test case input data for the problem.
     + **Output File**: Expected output to compare against.
   * The **App Server** does not store data but contains business logic to process submissions.
4. **Dependencies for Execution**:
   * Fetching the input and output files is necessary for the evaluation.
   * These files are large and stored in a **File Storage System** (not on the app server).
   * A diagram of a computer system

     Description automatically generatedFetching these files from storage takes time (~2 seconds), causing delays.
5. **Post-Evaluation Tasks**:
   * If the code is correct:
     + Mark the problem as solved for the user.
     + Update the user's **score**.
     + Recalculate the user's **rank** (e.g., **PSP**, **BSE** updates).
   * Store this updated information in a **MySQL Database**.
6. **Database Schema**:
   * **Problems Table**: Stores details about each problem:
     + Problem ID, title, description, created/updated timestamps.
   * **Problem Test Cases Table**:
     + Problem ID.
     + Path to the input file and expected output file for each test case.
7. **Handling File Storage**:
   * **Input/Output Files** are not stored in the database due to their large size.
   * Instead, the database stores paths (references) to these files.
   * These files are stored in:
     + An **object storage system** within the same data centre or
     + A **CDN** (Content Delivery Network).

## CDN vs. Local File Storage:

***Should the file path point to a CDN or local file storage system?***

**Current Scenario**:

* Test case input and output files are only accessed by the application servers during code submission and evaluation.
* These files are not shared with users to prevent misuse or cheating (e.g., gaming the system).

**Why Object Storage?**

* **Proximity**: Application servers and object storage are located in the same physical region (e.g., US East Coast).
* **Low Latency**: Fetching files from nearby object storage is faster and sufficient for internal operations.
* **No User Access**: Since users are not accessing the files, there is no need to cache them globally using a CDN.

**Why Not CDN?**

* **Unnecessary Overhead**: CDN is primarily designed for reducing latency by caching data closer to end-users. In this case, the files are accessed only by application servers, not by external users.
* **Local Optimization**: Object storage in the same building or region as the application servers already provides efficient access without the need for global caching.
* **Controlled Access**: CDN is unnecessary because test data is never shared publicly.

**Conclusion**:

* If both application servers and object storage are collocated (e.g., same data centre or region), using object storage directly is the most efficient solution.
* CDN would only be needed if the application servers accessing the files were distributed across multiple geographic locations, which is not the case here.

**Problem Statement and Solutions for Faster Code Submissions**

**Problem Overview**

1. **Current Flow**:
   * Problem ID is used to fetch the input and output file paths from the **Problem Test Cases Table** in the database.
   * Files are downloaded from **Object Storage**.
   * Code is executed with the input file, and the output is compared with the expected output file.
   * The process is slow, especially during the file download phase.
2. **Pain Points**:
   * **Slow File Fetching**: Downloading files from object storage is time-consuming (e.g., 600ms per file).
   * **User Perception**: Delays in code evaluation create a poor user experience, making the platform feel slow.

**Objective**

* Make code submission and evaluation faster while ensuring:
  + Consistency of cached data, especially during contests where test files might be updated in real time.

**Key Components**

1. **Database Tables**:
   * **Problems Table**: Contains metadata for problems (e.g., title, description, timestamps).
   * **Problem Test Cases Table**: Stores test cases, including:
     + Problem ID.
     + File paths for input and expected output.
     + Updated timestamps for the files.
   * **Users Table**: Stores user details (e.g., ID, name, score).
   * **User\_Problem Table**: Tracks user progress:
     + User ID.
     + Problem ID.
     + State (e.g., attempted, solved).
2. **Current Steps for Code Evaluation**:
   * Query the **Problem Test Cases Table** to get file paths based on the problem ID.
   * Download files from object storage.
   * Run the user's code, using the input file and comparing the output with the expected output file.

**Challenges**

1. **Latency**:
   * Downloading files from object storage introduces delays, making code submissions slow.
2. **Consistency**:
   * During a contest, test cases may be updated (e.g., correcting errors in the output file).
   * Cached data must reflect these updates immediately to prevent users from accessing outdated or incorrect files.

**Proposed Solutions**

**Caching**:

* Store frequently accessed files (input/output) in a cache for quicker retrieval.

**Final Problem Statement**

1. **How can we speed up the code submission and evaluation process?**
2. **How can we maintain data consistency when using caching mechanisms, especially during live contests?**

**Caching Approaches**

1. **Local Caching**:
   * Each app server caches test files locally.
   * Files are stored in the RAM or disk of the individual app server.
2. **Global Caching**:
   * A dedicated caching server (or cluster) stores all files centrally.
   * App servers fetch files from this global cache instead of object storage.

**Current Workflow**

1. **Fetching Metadata**:
   * After a problem is submitted, the system queries the database to fetch metadata, including paths for InputFile and Expected\_OutputFile.
2. **File Download**:
   * Files are retrieved from object storage based on the paths.
   * This step is slow due to the nature of object storage (files broken into chunks, stored across multiple locations).

**Requirements**

1. Speed up the process of accessing input and expected output files.
2. Maintain consistency when files are updated in object storage.

**Proposed Solutions**

**1. Time-to-Live (TTL) Based Caching**

* Cache files locally or externally for a fixed time before marking them as stale.
* **Challenges**:
  + **Low TTL**: Frequent invalidation (e.g., 1-minute TTL) leads to high cache misses, resulting in frequent slow fetches from object storage.
  + **High TTL**: Updates to files in object storage may not reflect in cache until the TTL expires, causing consistency issues.
* **Insights**:
  + TTL is not ideal for this scenario due to difficulty in balancing between high cache hit rates and timely invalidation.

**2. Local Caching on App Servers**

* Cache test files directly on each app server's local storage (RAM or disk).
* **Advantages**:
  + Faster access since files are fetched locally, avoiding network latency.
  + Decentralized approach reduces the load on a central caching system.
* **Challenges**:
  + **Consistency**:
    - Use file metadata (e.g., timestamps) to ensure cached files are up-to-date.
    - Files with mismatched timestamps are invalidated and re-fetched from object storage.
  + **Storage Limitations**:
    - RAM-based caching is fast but has limited space, leading to frequent evictions and cache misses.
    - Disk-based caching offers more storage but is slower.

**3. Global Caching in External Storage**

* A dedicated caching system that stores files centrally for all app servers.
* **Advantages**:
  + Centralized cache ensures consistency for all app servers.
  + Easier to update since only one location is involved.
* **Challenges**:
  + **Single Point of Failure (SPOF)**:
    - If the cache server crashes or becomes slow, it can degrade performance across the system.
    - Use replication to mitigate SPOF risks, but this adds complexity.
  + **Latency**:
    - Accessing files over the network is slower compared to local caching.

**Proposed Local Caching Solution**

1. **Fetching File Metadata**:
   * Each app server fetches the file metadata (input\_file\_path, output\_file\_path, and updated\_at timestamp) from the database.
2. **Constructing File Names**:
   * Cache files locally on each app server using file names that include the updated\_at timestamp.
     + Example: output\_<file\_path>\_<updated\_at>.
   * When files are updated in object storage:
     + The updated\_at value in the database changes.
     + The new file name includes the updated timestamp, which ensures the app server fetches the latest file.
     + Old cached files remain unused since the app server looks for the updated file name.
3. **Automatic Cache Invalidation**:
   * No need for TTL:
     + The change in updated\_at automatically invalidates stale files since the file name changes.
     + Files are fetched from object storage only if they are not found locally.
4. **Handling Requests**:
   * **First request after an update**:
     + The app server fetches the updated file from object storage.
   * **Subsequent requests**:
     + The file is found locally, making access faster.

**Challenges and Solutions**

1. **Duplicate Data Across App Servers**:
   * Every app server stores its own copy of the test files.
   * **Solution**:
     + Hard disk space on app servers is inexpensive and usually underutilized.
2. **Cleanup of Old Files**:
   * Multiple versions of files (with different timestamps) may accumulate on app servers.
   * **Solution**:
     + Use a **Cron job** to periodically clean up old, unused files.
     + Identify unused files based on the **last access time** rather than creation time.

**Challenges with Central Cache**

1. **Traffic Surge**:
   * Relying heavily on a central cache can lead to performance issues if there is a **sudden spike in traffic**.
   * A high volume of requests to the central cache can overwhelm it, causing slowdowns.
   * This can make all requests slow, as the cache server is under heavy load.
2. **Decentralized Caching**:
   * A decentralized approach may be more scalable, as it can distribute the load across multiple systems, reducing the risk of a single point of failure or overload.
   * Local caches (on app servers) could help manage traffic better.

**Easy vs Hard Test Cases**

1. **Easy Test Cases**:
   * These involve smaller files that are **easier to cache** and **faster to access**.
   * They can be quickly validated, without needing to check large files, making the system more efficient for these cases.
2. **Hard Test Cases**:
   * These are larger files that take more time to process and are **slower to cache**.
   * The system should prioritize correctness for easy test cases first and only handle large files (hard test cases) when necessary.
3. **File Structure**:
   * Every problem has two types of test data files:
     + **input\_01.txt** for easy test cases.
     + **input\_02.txt** for hard test cases (larger files).
     + Corresponding output files: **output\_01** and **output\_02**.
4. **Optimization Strategy**:
   * **Easy Test Cases**: Quickly validated, do not require accessing large files unless correctness is an issue.
   * **Hard Test Cases**: Only accessed if the solution passes the easy test cases but fails on correctness for larger cases.

## Caching the Rank List

**Problem:**

* **Scenario**: A gaming platform (like Dream11 or Scaler for students rank list) needs to compute a dynamic **rank list** based on users' scores.
* **Challenge**: Computing the rank list every time a request is made can be **expensive** and **slow**, as it requires sorting the user scores, which places a high load on the database.
* **Goal**: Reduce the load on the database and make the rank list more efficient to retrieve by caching it.

**Cache Options: Local vs. Global**

**Local Cache:**

* **Pros**:
  + **No single point of failure**: Load is distributed across multiple app servers.
  + **Distributed caching**: Each server handles its own cache, which can be more scalable in a distributed system.
* **Cons**:
  + **Increased DB Load**: If the cache is not available locally, the system queries the database, adding load to the DB.
  + **Stale Cache**: Every cache must refresh its data every 2-5 minutes, leading to many requests to the DB at short intervals.
  + **Increased complexity**: If the rank list needs to be refreshed on multiple servers, this requires a mechanism to handle cache synchronization.

**Global Cache:**

* **Pros**:
  + **Centralized Cache**: All servers fetch the rank list from a single global cache, reducing the DB load to just one query for the rank list update.
  + **Efficiency**: Only one query goes to the database every 2 minutes to refresh the global cache, reducing load on the database.
  + **Simplified Maintenance**: Centralized cache management reduces the need for synchronization across multiple app servers.
  + **Support for Sorted Sets**: Using technologies like Redis, the rank list can be stored as a sorted set, allowing for efficient retrieval and querying of specific ranks or ranges.
* **Cons**:
  + **Single Point of Failure**: If the global cache fails or becomes overloaded, it can affect all servers.
  + **Scaling Challenges**: If traffic spikes, the global cache (e.g., Redis) might face challenges in handling large amounts of data or high traffic.

**How to Refresh the Rank List Every 10 Minutes**

1. **Global Cache Strategy**:
   * Use **Redis** (or a similar tool) to store the rank list as a **sorted set**.
   * Every 10 minutes, the system queries the database once to compute the latest rank list and stores it in the global cache (Redis).
   * This approach ensures that the rank list is available for all app servers, reducing the number of queries to the database.
   * **TTL (Time-to-Live)**: Set a TTL on the cache, so it expires after 10 minutes, forcing a refresh.
2. **Local Cache Strategy**:
   * Each app server maintains a local cache and refreshes it every 2-5 minutes.
   * If the cache is empty, the server queries the database, which adds load to the system.
   * In case of a traffic surge, the load can be distributed across servers, but the database may still experience high traffic.

**Caching Technology (Redis):**

* Redis is an in-memory key-value store that supports advanced data structures like **sorted sets**, which are ideal for rank lists.
* **Advantages**:
  + Efficient for storing and retrieving rank lists.
  + Sorted sets allow for paginated retrieval (e.g., fetching ranks 10-30).
  + **ZADD** command in Redis allows adding users with their scores to the sorted set.
* **Global Cache**:
  + Redis as a global cache can handle the high traffic with the ability to scale horizontally by adding more Redis instances if needed.

## Cache Hit and Cache Miss:

* **Cache Hit**: When a requested value is found in the cache, it's called a **hit**. For example, if you're trying to fetch a user score from a cache and it's found, it's a hit.
* **Cache Miss**: When the requested value is not found in the cache, it's a **miss**. In this case, the system will fetch the data from the **database** instead.

**Cache Effectiveness:**

* **Cache is Useful When**: The cache improves performance only if the cache hit rate is high. If the system finds the requested data in the cache frequently, it reduces the time spent querying the database.
  + Example: If fetching from the cache takes **T1** time and fetching from the database takes **T2** time, then the total time spent in case of a miss (T1 + T2) increases latency.
  + If the cache miss rate is high, meaning the data is rarely found in the cache, it becomes less effective and may actually slow down the system because it adds an additional layer (cache lookup before hitting the DB).

**When to Cache:**

* **Good Cache Candidates**: Data that is frequently accessed or has repeated access patterns should be cached, such as frequently requested user scores or rank lists.
* **Avoid Caching**:
  + **Real-time Data**: Data that changes frequently (e.g., stock prices) should not be cached because the cache will miss frequently, and it would offer no performance benefit.
  + **Low Access Data**: Data that is accessed infrequently (e.g., Aadhar number lookup for many users) should also not be cached, especially if the cache size is small and the cache miss rate would be high.

**Cache Management Considerations:**

* **Cache Size**: You must decide the size of the cache and its eviction policy (how old or infrequently used data is removed to make space for new data).
* **Eviction Policy**: This helps in keeping the cache from becoming too large and inefficient by removing less frequently accessed data.

## Facebook’s Newsfeed on Profile Pages

**Problem Overview:**

* **Profile Page**:
  + Displays all posts made by a specific user (e.g., "Kaushik").
  + Includes posts created directly by the user.
  + May also include tagged posts or other activities (e.g., events).
  + For simplicity, focus on posts created by the user.
* **Newsfeed**:
  + Shows recent posts made by friends of the logged-in user.
  + Can also include posts from:
    - Pages the user follows.
    - Groups the user is a part of.
  + Posts are sorted or ranked based on relevance (e.g., recency, engagement).
  + Does not display very old posts (e.g., from 2 years ago).
  + Excludes ads for simplicity.

**Goal:**

* Design a system to **store** and **retrieve** the required data quickly so that:
  + Newsfeed and profile pages load fast.
  + Users have a smooth experience.

### Schema Design

**1. Users Table:**

* **Purpose**: Stores user information.
* **Columns**:
  + UserID: Primary Key (unique identifier for each user).
  + Name: User's name.
  + Other attributes like RelationshipStatus, etc.

**2. Posts Table:**

* **Purpose**: Stores posts created by users.
* **Columns**:
  + PostID: Primary Key (unique identifier for each post).
  + UserID: Foreign Key (references Users.UserID), indicates the post creator.
  + Content: Text content of the post.
  + AttachmentURL: Stores URL for attached media (e.g., images/videos).
  + CreatedAt: Timestamp of post creation.

**3. Friends Table:**

* **Purpose**: Tracks friendships between users.
* **Columns**:
  + User1ID: Foreign Key (references Users.UserID), one user in the friendship.
  + User2ID: Foreign Key (references Users.UserID), the other user in the friendship.
  + CreatedAt: Timestamp of when the friendship was established.
  + IsCloseFriend: Boolean to indicate close friendships (optional).

**4. UserLikes Table:**

* **Purpose**: Tracks which user likes which post.
* **Columns**:
  + UserID: Foreign Key (references Users.UserID).
  + PostID: Foreign Key (references Posts.PostID).
  + LikedAt: Timestamp of when the post was liked.

**Query Design:**

**1. Profile Page Query:**

* **Goal**: Retrieve all posts made by a specific user (e.g., Kaushik).
* **Steps**:
  + Query the Posts table to fetch posts where UserID matches the user.
  + Sort posts by CreatedAt in descending order to show the most recent posts first.
  + Implement pagination using LIMIT and OFFSET.

|  |
| --- |
| SELECT \*  FROM Posts  WHERE UserID = :KaushikID  ORDER BY CreatedAt DESC  LIMIT 10 OFFSET :offset; |

**2. Newsfeed Query:**

* **Goal**: Retrieve posts from friends of a user (e.g., UserID = 10), created within the last 30 days.
* **Steps**:
  1. Find all friends of the user from the Friends table.
  2. Join with the Posts table to retrieve posts created by these friends.
  3. Filter posts by timestamp (CreatedAt > now - 30 days).
  4. Sort posts by CreatedAt in descending order.
  5. Use LIMIT and OFFSET for pagination.

|  |
| --- |
| SELECT Posts.\*  FROM Friends  JOIN Posts  ON (Friends.User2ID = Posts.UserID OR  Friends.User1ID = Posts.UserID)  WHERE Friends.User1ID = :UserID OR  Friends.User2ID = :UserID  AND Posts.CreatedAt > NOW() - INTERVAL 30 DAY  ORDER BY Posts.CreatedAt DESC  LIMIT 10 OFFSET :offset; |

**Assumptions:** Queries assume all data resides on a single machine.

Given the massive scale of Facebook, with billions of users, each potentially generating thousands of posts and having thousands of friends, it's evident that storing all this information on a single machine's hard disk would be impractical. The sheer volume of data would far exceed the storage capacity of any single machine.

**Why Sharding is Needed:**

* Facebook scale involves billions of users, millions of posts, and large amounts of data that cannot fit into the storage of a single machine.
* Sharding helps distribute data across multiple machines.

**Sharding by User ID:**

* Users are distributed across multiple machines based on their UserID.
* Each shard contains:
  1. **User attributes**: Data from the Users table, e.g., name, relationship status.
  2. **Posts by the user**: All posts authored by the user.
  3. **Friends of the user**: IDs of the user's friends.
  4. **Optional Data**: Other lightweight associations (e.g., IDs of posts liked by the user).

**Data on a Shard (for a user, e.g., Kaushik):**

1. **User Attributes**: One row from the Users table for Kaushik.
2. **Posts Made by Kaushik**: Rows from the Posts table where UserID matches Kaushik's ID.
3. **Friend IDs**: List of Kaushik's friends stored as integers (e.g., 5000 friends = 20 KB).
4. **Exclusions**:
   * Posts made by Kaushik's friends are **not** stored on Kaushik's shard.
   * This avoids overwhelming the shard with unnecessary data.

**Querying with Sharded Data:**

**1. Profile Page Query:**

* **Goal**: Retrieve posts authored by Kaushik.
* **Execution**:
  + Go to the shard containing Kaushik.
  + Fetch all posts from Posts table where UserID = KaushikID.
  + Sort by CreatedAt in descending order for recent posts.
* **Performance**: Fast, as all required data resides on a single shard.

**2. Newsfeed Query:**

* **Goal**: Retrieve recent posts made by Kaushik's friends.
* **Steps**:
  1. **Fetch Friends IDs**:
     + Retrieve Kaushik's friend IDs from the Friends table on his shard.
  2. **Fetch Posts by Friends**:
     + For each friend ID, query the corresponding shard to retrieve their posts.
     + Filter posts by recency (CreatedAt > NOW() - INTERVAL 30 DAYS).
  3. **Aggregate Posts**:
     + Collect posts from all shards.
     + Sort by CreatedAt to prioritize the most recent posts.
* **Performance Challenge**:
  1. **Step 1 (Fetch Friends)**: Fast, as all friend IDs are on Kaushik's shard.
  2. **Step 2 (Fetch Posts)**: Potentially slow because:
     + Posts of friends are scattered across multiple shards.
     + Requires multiple queries to different shards.
     + Aggregation of results adds overhead.

**Problem:**

* **Newsfeed Generation is Slow**:
  + Fetching posts from all friends requires querying multiple shards.
  + High latency due to the distributed nature of data.
* **Impact**:
  + Increased load on Facebook's servers.
  + Poor user experience due to delays.

## Back-of-the-Envelope Estimation for Newsfeed Caching

**Problem Statement:**

* Newsfeed retrieval requires optimizing performance for billions of users.
* **Caching Strategy**:
  + Cache recent posts for quick access.
  + Assess feasibility of caching recent posts in terms of storage.

**Estimation Steps:**

1. **Daily Active Users (DAU)**:
   * **Assumption**: 1 billion daily active users.
   * **User Activity Distribution (Pareto Principle)**:
     + **80% (Passive Users)**: Only read posts.
     + **20% (Minimal Engagement)**: Like posts, minimal interactions.
     + **1% (Content Creators)**: Actively post.
2. **Number of Posts per Day**:
   * **Posting Users**: 1% of DAU → 10 million.
   * **Posts per User**: On average, 3-5 posts daily.
   * **Total Posts**: 10M × 5 = **50 million posts/day** (upper estimate).
3. **Post Size Breakdown**:
   * **Post Components**:
     + **Text**: Average 100 characters.
     + **CDN URL**: 100–200 characters (for multimedia).
   * **Total Size**: ~500 bytes/post (text + CDN URL).
   * **Encoding**:
     + ASCII: 1 byte/character.
     + Unicode: 2 bytes/character.
   * **Storage Size**:
     + **Daily Storage**: 50M posts × 500 bytes = **25 GB/day**.
4. **Storage for Last 30 Days**:
   * **Monthly Storage**: 25 GB × 30 days = **750 GB**.

**Feasibility Analysis:**

1. **Can Data Be Stored in RAM?**
   * **Modern RAM Capacity**: Generally up to ~512 GB (high-end servers).
   * **Conclusion**: **Not feasible** to store 750 GB entirely in RAM.
2. **Can Data Be Stored on Disk?**
   * **Disk Storage**: Modern disks can easily support **750 GB**.
   * **Single MySQL Instance**:
     + **Possible**, but requires high-end hardware:
       - 128 GB RAM, SSD storage, 3 GHz+ octa-core CPU.
     + **Performance Concerns**:
       - Latency increases exponentially as MySQL approaches its storage limits.

## Caching Strategy for Newsfeed

**Problem:**

* Generating a newsfeed is computationally intensive due to large-scale data and frequent updates.
* The challenge is optimizing performance while managing storage and scalability.

**Solution: Recent Posts Database (RPDB)**

1. **Two-Tiered Database Approach**:
   * **User Database (UDB)**: Stores all user data and historical posts.
   * **Recent Posts Database (RPDB)**:
     + Stores only recent posts (e.g., last 30 days).
     + Acts as a cache layer for the newsfeed.
     + Contains a single table with columns:
       - **Post ID**
       - **User ID**
       - **Timestamp**
       - **Content**
       - **File URL (for multimedia)**
2. **Feasibility of RPDB**:
   * **Assumption**: Recent posts (~750 GB for 30 days) can fit in a single machine with high-end specifications.
   * **Storage**:
     + Data stored on disk, not RAM.
     + Machine supports multiple replicas for availability and load balancing.
   * **Why It Works**:
     + The size of recent posts (30 days) is small enough to fit in one machine.
     + Newsfeed consistency is not critical; slight delays or missing posts are acceptable.

**Workflow:**

1. **Fetching Newsfeed**:
   * Retrieve the friend list from **UDB**.
   * Query **RPDB** for posts by those friends:

|  |
| --- |
| SELECT \* FROM posts  WHERE user\_id IN (list\_of\_friends)  ORDER BY timestamp DESC  LIMIT 25 OFFSET 0; |

1. **Updating RPDB**:
   * When a user posts:
     + Update **UDB** with the new post.
     + Simultaneously write to **RPDB**:
       - Include the post ID, user ID, timestamp, and content.
2. **Data Expiry**:
   * Scheduled daily jobs remove posts older than 30 days from **RPDB**.
3. **Cache Invalidation**:
   * Not required for posts since they are immutable.
   * Updates (like image deletion) can be handled by scheduled tasks, not immediate actions.

**Key Insights:**

1. **Caching Redefined**:
   * Cache does not have to be in RAM or a key-value store.
   * RPDB serves as a database cache layer optimized for recent posts.
2. **Newsfeed Design Assumptions**:
   * Only recent posts are required for newsfeed generation.
   * Posts beyond 30 days are irrelevant for typical use cases.
3. **Consistency vs. Availability**:
   * Availability is prioritized over strict consistency since slight delays in post updates are acceptable.
4. **Scalable Architecture**:
   * RPDB ensures scalability without requiring complex sharding for recent posts.

## Managing Load and Refresh for Newsfeed

**Problem 1: Handling Load on Recent Posts Database (RPDB)**

1. **Challenge**:
   * Newsfeed is a frequently used feature.
   * High traffic could overload the RPDB.
2. **Solution**: Use **Replicas**.
   * **What are replicas?**
     + Multiple copies of the RPDB to distribute the load.
     + Each replica contains the same data, ensuring redundancy.
   * **Advantages**:
     + Requests are distributed evenly across replicas using a load balancer or round-robin mechanism.
     + Reduces the load on individual machines.
   * **Master-Slave Replication**:
     + Writes go to a master database.
     + Updates propagate asynchronously to replicas (slaves).
     + Slight delays in replication are acceptable as consistency isn’t critical for the newsfeed.
3. **Why Sharding Isn’t Ideal**:
   * Sharding RPDB by time (e.g., 0–15 days and 16–30 days) can lead to **load skew**:
     + Most requests will hit the shard with the most recent data.
   * Additional complexity arises if multiple shards need to be queried.

**Problem 2: Refreshing Newsfeed Quickly**

1. **Challenge**:
   * The newsfeed should appear updated without delays or user noticing work in progress.
2. **Solution 1: Delta Fetch**:
   * **What is Delta Fetch?**
     + A mechanism where the client fetches only new posts since the last update.
   * **How it works**:
     + The client sends a **timestamp** of the last refresh with the request.
     + RPDB query retrieves posts made after this timestamp: