HLD Popular Interview Questions

Contents

[1. Distributed ID Generator Problem 2](#_Toc184913601)

[Thought Process 4](#_Toc184913602)

[Twitter Snowflake ID Algorithm and Distributed Unique ID Generation 5](#_Toc184913603)

[How it works - Twitter Snowflake Algorithm 8](#_Toc184913604)

[Handling Edge Cases 10](#_Toc184913605)

[2. Rate Limier 13](#_Toc184913606)

[Distributed Denial of Service (DDoS) 13](#_Toc184913607)

[Rate Limiting 14](#_Toc184913608)

[Rate Limiting and DDoS Mitigation 15](#_Toc184913609)

[Rate Limiting Algorithm 17](#_Toc184913610)

[3. URL Shortener Design 21](#_Toc184913611)

# Distributed ID Generator Problem

**Problem Statement**

* **Objective:** Design a system to generate unique and incremental IDs in a distributed setup.
* **Key Requirements:**
  1. **Uniqueness:** Each ID must be unique across all machines in the system.
  2. **Incrementality:** IDs must reflect temporal order:
     + If event A occurs before event B, ID of A < ID of B.
     + For strings, use lexicographical order.
     + Incrementality does not require strict consecutive values (e.g., ID 5 followed by 6).
  3. **Efficiency:** The solution should scale with increasing traffic and number of machines.

**Challenges**

1. **Distributed Database Setup:**
   * Data is sharded across multiple machines.
   * Auto-increment IDs (like in relational databases) are not directly feasible due to distributed architecture.
2. **Centralized ID Generation:**
   * A single machine generating IDs introduces:
     + **Bottleneck:** All requests funnel through one machine.
     + **Single Point of Failure:** Failure of this machine disrupts ID generation.
   * Suitable only for low-frequency use cases (e.g., infrequent image uploads).

A diagram of a computer network

Description automatically generated

1. **Clock Synchronization Assumption:**
   * System assumes synchronized clocks across all machines.
   * NTP (Network Time Protocol) can help achieve this.

**Additional Considerations**

* **Performance:**
  + Integer IDs are faster to compare than string-based IDs.
  + 64-bit integers (e.g., long long int) are efficient for storage and comparison.
* **Design Complexity:**
  + Ensure simplicity in implementation while adhering to scalability and fault-tolerance requirements.
* **Failure Handling:**
  + Backup mechanisms for critical components to minimize downtime.

## Thought Process

* Begin by brainstorming solutions within the constraints:
  1. ID generation must be distributed to avoid bottlenecks.
  2. Time-based ordering or machine identifiers might help maintain incrementality and uniqueness.
* Ensure that the design leverages:
  1. Synchronization protocols for time consistency.
  2. Distributed mechanisms to avoid dependency on a single machine.

1. **Initial Suggestions:**
   * **Using Timestamp as ID:** Timestamp is incremental but not unique (two messages can share the same timestamp).
   * **UUID/GUID:** Unique but not incremental; cannot satisfy the ordering requirement.
   * **Shard ID + Counter + Timestamp:**
     + Provides uniqueness but violates incrementality (e.g., larger Shard ID but earlier timestamp).
   * **Partitioning IDs:** Machines generate IDs from pre-assigned ranges (e.g., multiples of 3, 3x+1, 3x+2):
     + Ensures uniqueness but not incrementality when requests are routed randomly.
     + Enforcing incrementality requires a central coordinator, introducing bottlenecks.
2. **Timestamp Granularity:**
   * Fine-grained timestamps (e.g., nanoseconds) reduce collision probability but do not guarantee uniqueness.
   * Cannot rely on network communication between machines for collision resolution due to latency concerns.

**Insights and Key Decisions**

* **Timestamp Usage:**
  + Use timestamp as the foundation since it is inherently incremental.
  + Append additional information to ensure uniqueness.
* **Placement of Timestamp:**
  + Timestamp must be the **most significant part** of the ID:
    - Guarantees that if timestamp T1 < T2, then ID1 < ID2.
    - Placing timestamp at the start ensures correct ordering during comparisons.
* **Append Strategy:**
  + Append machine-specific or random data after the timestamp to ensure uniqueness.
  + Avoid appending data before the timestamp, as this disrupts the incremental property.
* **Next Steps:**
  + Evaluate schemes for appending data that maintain uniqueness without violating incrementality.
  + Discuss benefits of incremental IDs (to be covered later in the lecture).

## Twitter Snowflake ID Algorithm and Distributed Unique ID Generation

**Problem Statement**

The goal is to generate unique and incremental IDs in a distributed system with the following constraints:

1. Multiple machines are generating IDs independently.
2. No inter-machine communication for synchronization.
3. IDs must be unique and ideally incremental.

**Key Components for Unique and Incremental IDs**

The algorithm ensures uniqueness and incremental properties by combining:

1. **Timestamp:** To ensure chronological order.
2. **Machine ID:** To distinguish between different machines or data centres.
3. **Sequence ID:** To handle multiple requests on the same machine at the same timestamp.

**Snowflake Algorithm Overview**

Twitter's Snowflake algorithm uses a 64-bit integer to encode the ID with specific bit allocations:

1. **42 bits for Timestamp:**
   * Provides coverage up to a large time range (e.g., till 2050).
   * Ensures chronological order.
2. **10 bits for Machine ID:**
   * Split into:
     + 4 bits for Data Centre ID: Supports up to 15 data centres.
     + 6 bits for Machine/Shard ID: Supports up to 64 machines per data centre.
   * Differentiates IDs generated by different machines or shards.
3. **12 bits for Sequence ID:**
   * Allows up to 4095 unique IDs per machine per timestamp.
   * Handles multiple requests within the same timestamp on the same machine.

**Workflow of ID Generation**

1. **Timestamp:**
   * The first 42 bits encode the current timestamp.
   * Ensures IDs are ordered chronologically.
2. **Machine ID:**
   * Encodes the Data Centre ID and Machine/Shard ID.
   * Differentiates machines to prevent conflicts across machines.
3. **Sequence ID:**
   * Uses an atomic integer to generate unique sequence numbers for requests arriving at the same timestamp.
   * Rolls over (e.g., modulo 4096) if more than 4095 requests occur at the same timestamp.

**Incremental Properties of IDs**

* **Chronological Ordering:**
  + The higher timestamp guarantees a larger ID.
  + If two timestamps are equal, the Machine ID and Sequence ID are used to ensure uniqueness.
* **Bit-wise Comparison:**
  + When comparing two IDs, the 42-bit timestamp ensures the correct ordering unless timestamps are equal.
  + If timestamps are identical, the 10-bit Machine ID or 12-bit Sequence ID will differentiate the IDs.

**Multi-threading Considerations**

* **Atomic Integer:**
  + Ensures that sequence numbers are unique within the same timestamp.
  + Handles concurrent requests by incrementing the value atomically.
* **Locks and Concurrency:**
  + Uses internal locks to manage increments safely across threads.

**Conflict Handling**

* **Rare Conflicts:**
  + Conflicts can occur if more than 4095 messages are received at the same timestamp on the same machine.
  + Probability of such conflicts is negligible in most real-world scenarios.

**Adaptability of Bit Allocation**

* **Adjusting Timestamp Bits:**
  + If more granular timestamps are required, additional bits can be allocated to the timestamp.
* **Adjusting Machine ID Bits:**
  + If more machines or data centres are needed, bits can be reallocated from the Sequence ID.
* **Adjusting Sequence ID Bits:**
  + For higher throughput, more bits can be allocated to the Sequence ID.

**Summary of Bit Allocation**

* **42 bits:** Timestamp (chronological order).
* **4 bits:** Data Centre ID (up to 15 data centres).
* **6 bits:** Machine/Shard ID (up to 64 machines per data centre).
* **12 bits:** Sequence ID (up to 4095 IDs per machine per timestamp).

**Key Benefits of the Algorithm**

1. **Scalability:** Works efficiently in distributed environments with multiple machines.
2. **Uniqueness:** Guarantees unique IDs even in high-concurrency scenarios.
3. **Incrementality:** Ensures IDs are incremental when timestamps differ.
4. **Flexibility:** Bit allocations can be adjusted based on system requirements.

## How it works - Twitter Snowflake Algorithm

A 64-bit ID is constructed using the following components:

1. **Timestamp (42 bits):**
   * Represents the time when the ID is generated.
   * Covers time ranges sufficiently far into the future (e.g., 2015–2050).
   * Ensures IDs are ordered chronologically.
2. **Data Center and Machine ID (10 bits):**
   * **4 bits** for Data Center ID: Supports up to 15 data centers.
   * **6 bits** for Machine/Sharding ID: Supports up to 64 machines per data center.
   * Differentiates IDs generated in different physical locations or machines.
3. **Sequence ID (12 bits):**
   * Counts the number of messages generated with the same timestamp on the same machine.
   * Values range from 0 to 4095 (12 bits), allowing for up to 4095 unique IDs per millisecond.

**How It Works:**

* **Ensuring Uniqueness:**
  + Combination of Timestamp, Data Center + Machine ID, and Sequence ID guarantees uniqueness:
    1. If two messages are generated on the same machine at the same time, Sequence ID ensures different values.
    2. If two messages have the same timestamp but come from different machines, Data Center + Machine ID ensures differentiation.
* **Ensuring Incremental Order:**
  + **Primary Order:** Timestamp ensures IDs generated later have higher values.
  + **Secondary Order:** If timestamps are the same:
    1. **Different Machines:** Data Center + Machine ID ensures no clash.
    2. **Same Machine:** Sequence ID ensures monotonic increments.

**Concurrency Handling (Sequence ID Generation):**

* Multi-threading is handled via atomic operations:
  + Each thread accesses a shared integer (e.g., AtomicInteger in Java) to generate a unique sequence ID.
  + Atomic operations ensure no two threads receive the same sequence number at the same timestamp.

**Customization:**

* The bit allocation among Timestamp, Data Center ID, Machine ID, and Sequence ID can be adjusted based on:
  + Expected number of data centers or machines.
  + Granularity of timestamps.
  + Maximum number of messages per millisecond.

**Key Properties of Snowflake Algorithm:**

1. **Distributed Generation:**
   * Machines can independently generate IDs without intercommunication.
2. **Uniqueness:**
   * No conflicts across timestamps, machines, or parallel threads.
3. **Incremental:**
   * Chronological ordering ensured by timestamp bits.

**Edge Case Considerations:**

* **Message Overload:** If a machine receives more than 4095 messages in a millisecond:
  + Sequence ID will overflow, potentially causing a conflict.
  + This is a rare scenario given the improbability of such high throughput per machine.
* **Concurrent Threads:**
  + Handled using locks or atomic data structures (e.g., atomic integers) to avoid sequence ID clashes within the same machine.

**Comparison with UUID:**

* **UUID:**
  + Uses hashing, which provides uniqueness but lacks chronological order.
* **Snowflake:**
  + Provides both uniqueness and incremental IDs, making it suitable for systems requiring order-preserving identifiers.

## Handling Edge Cases

1. **Same Timestamp**:
   * If multiple IDs are generated at the same timestamp, the **Sequence ID** ensures uniqueness.
   * No priority exists; the order within the same timestamp doesn’t matter.
2. **Overflow of Sequence ID**:
   * Rare, as 4095 IDs per millisecond is a high limit.
   * If overflow occurs, the system can wait for the next millisecond (rate-limiting).
3. **Message Failure**:
   * If a request fails, the Sequence ID is not reused; gaps in the ID sequence are acceptable.
   * Reduces overhead and complexity.

**Advantages of Incremental IDs**

1. **Database Indexing**:
   * Many databases automatically index IDs, especially primary keys.
   * Queries like "find all entries where ID > X" become efficient.
   * Delta fetch (e.g., fetching tweets or messages) benefits from using IDs rather than timestamps.
2. **Chronological Ordering**:
   * IDs naturally sort data by time when timestamp forms the most significant bits.
3. **Reduced Overhead**:
   * Distributed ID generation avoids bottlenecks and central system failures.

**Performance of Snowflake ID Generation**

* **Efficient Operations**:
  + Data center ID and machine ID are constants, fetched at startup.
  + Timestamp retrieval and bit manipulation (shifting) are lightweight.
  + Sequence ID increment is a simple atomic operation.
* **Processing Time**:
  + Operations like timestamp retrieval and sequence increment require only a few CPU cycles.

**Example of Snowflake ID Generation**

* Consider two timestamps:
  + Timestamp: **998**, Sequence ID: **2** → ID: **9982**.
  + Timestamp: **999**, Sequence ID: **1** → ID: **9991**.
* The ID at timestamp **999** is larger than the ID at timestamp **998**.

**Security Concerns**

* Guessability of IDs:
  + Relevant if IDs expose sensitive information.
  + For non-sensitive systems like tweets, authentication checks prevent unauthorized access.

**Conclusion**

* **Incremental IDs** ensure efficient querying, natural chronological ordering, and high throughput in distributed systems.
* The **Snowflake algorithm** balances uniqueness, scalability, and performance without centralized dependency.

# Rate Limier

## Distributed Denial of Service (DDoS)

**DDoS Attack:**

1. **Definition:**
   * A Distributed Denial of Service (DDoS) attack overwhelms a target system (e.g., website) with a high volume of artificial traffic, making it inaccessible or very slow for legitimate users.
   * Can cause machines to fail or degrade the user experience significantly.
2. **Types of DDoS Attacks:**
   * **General Traffic Flood:** Overwhelms servers by sending a high volume of requests.
   * **Resource-Specific Attack:** Targets expensive operations like:
     + Image uploads.
     + Code submissions (e.g., on a coding platform like Scalar).
3. **Motivations Behind DDoS:**
   * Sabotage.
   * Extortion through ransom demands.

**Need for Protection Against DDoS:**

1. **Impact of DDoS:**
   * Real users face degraded performance or service outages.
   * Servers get overwhelmed, leading to failure.
2. **Solution: Rate Limiting**
   * Restricts the frequency and volume of requests from:
     + Specific IP addresses.
     + User IDs.
     + Session IDs.
     + Other identifiers (e.g., geographical location).

## Rate Limiting

1. **Purpose:**
   * Prevent abuse of server resources.
   * Mitigate the impact of DDoS attacks by limiting excessive or anomalous request patterns.
2. **Mechanism:**
   * Define rules like:
     + No more than 5 requests per second per IP address.
     + Limit logged-in users to 1 code submission per second, or 100 submissions per hour.
3. **Implementation Considerations:**
   * Rules should strike a balance:
     + Accommodate legitimate user behavior.
     + Restrict potential attackers.
   * Should be dynamic and configurable:
     + Based on traffic patterns.
     + Identifying and adapting to new attacker behaviors.
4. **Rate Limiting Challenges:**
   * Distributed attacks can originate from multiple IP addresses.
   * Requires identification of attack patterns and crafting appropriate rules:
     + E.g., session IDs, user IDs, location clusters.

**Rate Limiter Design:**

1. **Problem Statement:**
   * Build a rate limiter to enforce rules like:
     + Restrict requests from a specific IP address or user ID.
     + Limit resource usage per session.
2. **Key Features of a Rate Limiter:**
   * Allow users to define request limits (e.g., X requests per second/minute/hour).
   * Efficiently track and enforce limits in real-time.
   * Handle a large volume of requests while maintaining performance.
3. **Implementation Challenges:**
   * Efficient data storage and lookup to track request counts.
   * Scalability to handle high traffic without becoming a bottleneck.
   * Ensuring correctness even under heavy load or distributed scenarios.

## Rate Limiting and DDoS Mitigation

**Introduction to DDoS Attacks**

* **Definition**: Distributed Denial of Service (DDoS) attack overwhelms a website with excessive traffic to degrade user experience or crash systems.
* **Examples**:
  + Overloading websites with artificial requests during peak traffic hours.
  + Exploiting expensive operations, such as image uploads or code evaluations, to strain server resources.
* **Motivation for attackers**: Some attackers demand ransom to stop the attack.

**Need for Rate Limiting**

* **Purpose**:
  + Mitigate DDoS attacks by restricting traffic patterns indicative of malicious behaviour.
  + Ensure fair resource usage among legitimate users.
* **Examples of Rules**:
  + Limit requests per IP address (e.g., 10 requests per 60 seconds).
  + Restrict actions per user (e.g., 10 code submissions per minute).

**Where to Implement Rate Limiting**

* **Possible Locations**:
  + **Client-side**: For user-facing restrictions (e.g., CAPTCHA).
  + **Load Balancer**: For lightweight request filtering before hitting application servers.
  + **API Gateway**: Suitable for more advanced rules with application logic.
  + **Server-side**: For deep logic related to application-specific operations.

**Challenges in Implementing Rate Limiting**

1. **Efficient Request Tracking**:
   * Need to process large volumes of traffic without excessive memory or CPU usage.
2. **Accuracy vs. Efficiency**:
   * Approximations may suffice if they can effectively detect malicious patterns.

**Basic Approaches to Rate Limiting**

1. **Full Request Logging**
   * Maintain a log of every request with timestamps.
   * **Drawbacks**:
     + High memory usage.
     + Expensive write operations under heavy traffic.
2. **Bucket-Based Hashing**
   * **Concept**: Divide time into fixed buckets (e.g., 60 seconds). Track the number of requests per bucket.
   * **Implementation**:
     + Use a hashmap keyed by IP address with values representing request counts.
     + At the start of each bucket, reset the hashmap.
   * **Code Outline**:
     + Maintain a hashmap (H) where H[IP] = request\_count.
     + On each request:

|  |
| --- |
| if H[IP] >= MAX\_LIMIT:      reject\_request()  else:      H[IP] += 1 |

* + - Reset hashmap every bucket using a background thread.

|  |
| --- |
| Every 60 seconds:      H = new hashmap |

* + **Advantages**: **Lightweight** storage; reduced writes.
  + **Drawbacks**:
    - Boundary issues: **Requests** at the end of one bucket and start of the next may exceed limits (e.g., 18 requests in less than 60 seconds).
    - Not fully **accurate**.

**Advanced Considerations**

* **TTL (Time to Live)**: Using TTL-based expiry in storage systems like Redis may not be suitable if keys are frequently updated, as TTL resets on update.
* **Token Bucket Algorithm**: Mentioned but not elaborated; a method to manage request limits dynamically.
* **Action Upon Violation**:
  + Block user/IP.
  + Implement CAPTCHA or similar mechanisms for additional verification.
  + Return HTTP 429 (Too Many Requests) status code for rejected requests.

## Rate Limiting Algorithm

**Problem Description**

* **Scenario**: An IP address sends requests at specific timestamps.
  + Example:
    - Requests at 51, 52, 53, ..., 58, 59 (8 requests in 10 seconds).
    - More requests at 61, 62, 63, ..., 71, 72.
* **Rule**: Maximum 10 requests are allowed per 60-second window.

**Issue**:

* Using the current algorithm:
  + Requests from 51 to 59 are valid (within the limit).
  + At 61, the count resets to 0.
  + Requests from 61 to 70 are allowed even though they exceed the limit within the rolling 60-second window.

**Current Approach**

1. **Map-Based Count**:
   * Maintain a map of IP addresses to their request counts.
   * Count is tracked for 60-second windows.
   * If the count exceeds 10, the request is rejected.
2. **Resetting Count**:
   * At the start of every new window (e.g., t = 61), reset all counts to zero.
   * Alternatively, delete the map and create a new one for the current window.

**Problem**:

* **Boundary Issue**: Requests near the boundary of two windows can be miscounted, allowing more requests than the allowed limit.

**Proposed Solution: Sliding Window Approach**

1. **Two Maps**:
   * Maintain two maps:
     + **Current Map**: Tracks requests in the ongoing 60-second window.
     + **Previous Map**: Tracks requests in the previous 60-second window.
2. **Weighted Count**:
   * At any timestamp t, determine the total count of requests:
     + Exact count from the **current map**.
     + Weighted count from the **previous map**, based on the overlap duration.
   * Formula for weighted count:
     + If x seconds of the previous window are still relevant:
       - Requests from the previous map are scaled as x / 60.
     + Total count = Current map count + (x / 60) \* Previous map count.
3. **Window Transition**:
   * Every 60 seconds:
     + Move the current map to the previous map.
     + Clear the previous map and create a new current map.

**Example Calculation**

* **Scenario**:
  + Current timestamp: t = 80 seconds.
  + **Requests**:
    - Current map: 5 requests from IP 10.20.30.40.
    - Previous map: 8 requests from the same IP during the last window.
* **Weighted Count**:
  + Current window: Full 20 seconds.
  + Previous window: Relevant for 40 seconds.
  + Calculate:
    - Current count = 5.
    - Previous count = (40 / 60) \* 8 = 16 / 3 ≈ 5.33.
    - Total count = 5 + 5.33 ≈ 10.33.
* **Decision**:
  + Total count exceeds the limit of 10 requests.
  + Reject new requests.

**Advantages of Sliding Window**

* **Accuracy**: Provides a more precise count by considering requests near the boundary.
* **Efficiency**:
  + Two maps are lightweight compared to storing all timestamps.
  + Avoids expensive computations like iterating over all requests.
* **Flexibility**: Balances accuracy with simplicity by assuming even distribution of requests within the previous window.

**Key Insights**

1. **Sliding Window Mechanism**:
   * Current and previous windows overlap, ensuring smoother transitions.
   * Weighted contribution of the previous window decreases as time progresses.
2. **Limitations**:
   * Assumes uniform distribution of requests within the previous window.
   * Not as precise as tracking individual timestamps but significantly more efficient.
3. **Implementation Trade-Off**:
   * Lightweight and fast read operations.
   * Slightly less accurate but sufficient for most rate-limiting needs.

# URL Shortener Design

**Introduction**

* URL shorteners simplify long URLs into shorter, manageable links (e.g., facebook.com/... becomes bit.ly/XYZ).
* They maintain mappings between shortened URLs and original URLs for redirection.
* Difficulty level is moderate: simpler than a messenger system.

**Core Functionality**

1. **Input and Mapping**:
   * Input: Long URL (e.g., facebook.com/...).
   * Output: Shortened URL (e.g., bit.ly/XYZ).
   * Store mappings in a data structure or database.
2. **Redirection**:
   * When bit.ly/XYZ is accessed, lookup XYZ in the database.
   * Redirect to the original URL (e.g., facebook.com/...).

**Implementation Details**

1. **Storage**:
   * Use a key-value storage system (e.g., SQL database with two columns):
     + Column 1: Shortened URL.
     + Column 2: Original URL.
2. **Hash Generation**:
   * Generate a random, unique string as the shortened URL key:
     + Example: Random 6-character string (e.g., CZKERT).
     + Algorithm:
       - Generate random characters using rand() % 26 + 'A'.
       - Repeat for 6 characters.
       - Check database for collision; regenerate if needed.
   * Alternatively, use MD5 or hash of a timestamp.

**Additional Features**

1. **Custom Shortened URLs**:
   * Allow users to define custom shortened strings (e.g., bit.ly/JamesBond).
   * Ensure uniqueness by checking the database before creating the entry.
2. **Analytics**:
   * Track the number of clicks per shortened URL.
   * Use an additional table:
     + Columns: Shortened URL, Count.
   * For high-traffic scenarios:
     + Store counts in a faster, in-memory store like Redis.
     + Periodically sync with the SQL database.

**Request Handling**

1. **Shortened URL Access**:
   * Access bit.ly/XYZ sends a request to bit.ly servers with XYZ as the argument.
   * Server looks up XYZ in the database to find the original URL.
   * Responds with an HTTP redirect to the original URL.
2. **Server Response**:
   * HTTP Response includes a redirect directive.
   * Can also include options for blocking or rejecting a request.

**Conclusion**

* The URL shortener system is fundamentally a simple key-value mapping problem.
* Scalable design can include additional features like analytics and custom URLs.
* Storage and hash generation strategies ensure uniqueness and efficiency.