CH7 Design A Unique ID Generator IN Distributed Systems

What is a unique ID generator?

A unique ID generator is a tool or mechanism used to create unique identifiers or codes that can distinguish individual records, transactions, or elements within a system. These identifiers are crucial for ensuring that each item or entity within a database or application is uniquely and consistently recognized, without overlap or duplication.

Key Characteristics of Unique IDs:

- **Uniqueness**: Each generated ID must be distinct from all others within the relevant scope (e.g., within a database or across a distributed system).
- **Non-sequential**: IDs are often non-sequential to prevent predictability, which can be important for security reasons.
- **Fixed Length**: Many systems require IDs to be of a fixed length, which simplifies storage and processing.

Why do we need a unique ID generator?

A unique ID generator is essential in many computing and data management scenarios for several key reasons:

1. **Uniqueness and Identity**: Unique IDs provide a way to uniquely identify each record or entity in a system. This is crucial for differentiating between items that might otherwise be indistinguishable based on their attributes alone.
2. **Data Integrity**: In databases, unique IDs help maintain data integrity by ensuring that each entry can be accurately referenced and updated without

ambiguity. They are often used as primary keys in relational databases to establish and enforce relationships between tables.

3. **Scalability and Distribution**: **Unique IDs are vital in distributed systems** where data is spread across multiple servers or locations. They help ensure that data can be integrated and synchronized across different parts of the system without conflicts, even when the system scales or extends across geographical boundaries.
4. **Security**: Non-sequential or randomly generated unique IDs can enhance security by making it harder to predict the ID of a specific record, thereby preventing unauthorized data access.
5. **Traceability and Logging**: In event logging, unique IDs can track specific events or transactions throughout a system's workflow. This traceability is invaluable for debugging, monitoring, and auditing system behavior.
6. **Concurrency**: In concurrent computing, where multiple processes or threads access and modify data simultaneously, unique IDs help manage and coordinate these operations safely and effectively.

Unique ID generators support system robustness and functionality by providing these essential services, making them foundational components in modern software architecture and database design.

How do we design a unique ID generator in distributed systems ?

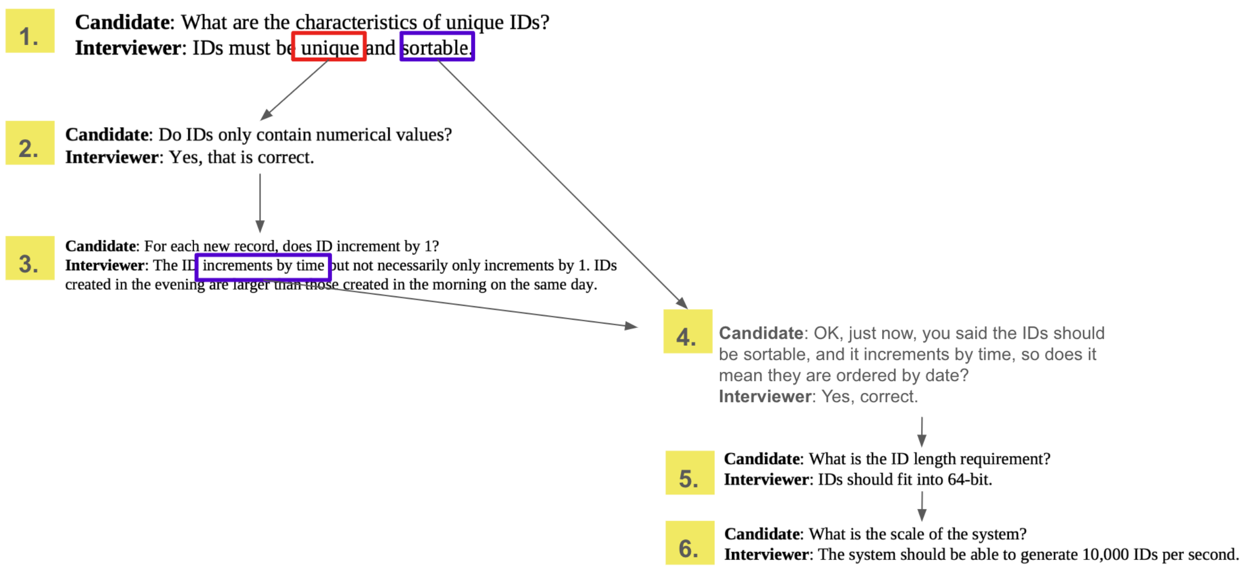
First though

```
CREATE TABLE Users (
    id INT AUTO_INCREMENT,
    username VARCHAR(255) NOT NULL,
    email VARCHAR(255) NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (id)
);
```

auto_increment does not work in a distributed environment because:

- a single database server is not large enough
- generating unique IDs across multiple databases with minimal delay is challenging

Step 1 - Understand the problem and establish design scope



1. **Candidate**: What are the characteristics of unique IDs?
Interviewer: IDs must be unique and sortable.

2. **Candidate**: Do IDs only contain numerical values?
Interviewer: Yes, that is correct.

3. **Candidate**: For each new record, does ID increment by 1?
Interviewer: The ID increments by time but not necessarily only increments by 1. IDs created in the evening are larger than those created in the morning on the same day.

4. **Candidate**: OK, just now, you said the IDs should be sortable, and it increments by time, so does it mean they are ordered by date?
Interviewer: Yes, correct.

5. **Candidate**: What is the ID length requirement?
Interviewer: IDs should fit into 64-bit.

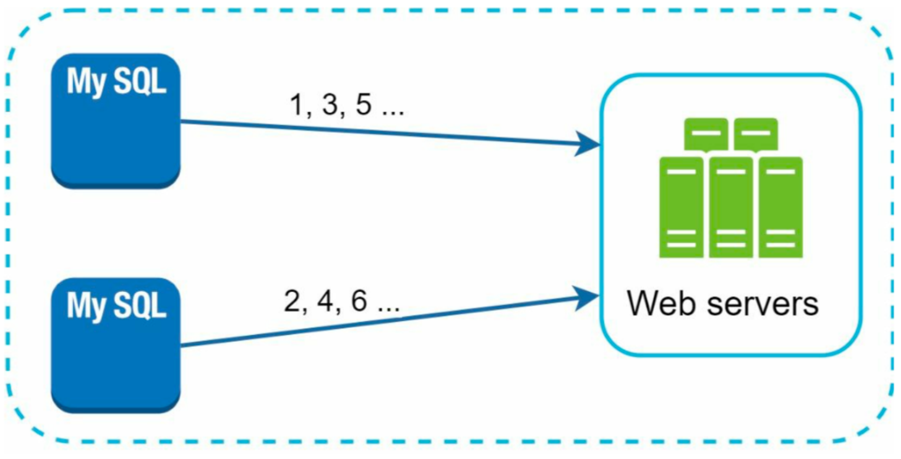
6. **Candidate**: What is the scale of the system?
Interviewer: The system should be able to generate 10,000 IDs per second.

Requirements:

- IDs must be unique.
- IDs are numerical values only.
- IDs fit into 64-bit.
- IDs are ordered by date.
- Ability to generate over 10,000 unique IDs per second.

Step 2 - Propose high-level design and get buy-in

1. **Multi-master replication**



The diagram illustrates a multi-master replication approach using an auto-increment feature in database management, specifically within the context of MySQL databases, to generate unique identifiers (IDs).

How does it work?

- **Multi-master Replication Setup**: The diagram shows two MySQL servers configured in a multi-master replication setup. This configuration allows each server to independently generate IDs and serve data, improving redundancy and availability.
- **ID Generation Strategy**: Each MySQL server generates unique IDs using the auto-increment feature. The servers are set to increment IDs by a step value `k`, which is the number of servers. In this setup:
    - The first server generates IDs like 1, 3, 5, etc.
    - The second server generates IDs like 2, 4, 6, etc.
- **Web Servers**: The web servers in the diagram are likely consuming the data from both MySQL servers, depending on their load balancing or data requirement setups.

Drawbacks:

- **Hard to Scale with Multiple Data Centers**: When deploying this architecture across multiple data centers, synchronization and latency issues can complicate the ID generation process, making it less efficient.

- **IDs Do Not Go Up with Time Across Multiple Servers**: Since each server generates IDs independently based on the auto-increment set up, IDs from different servers may not sequentially align with the time when the records are created. This can be problematic for applications requiring temporal data consistency across servers.
- **Does Not Scale Well When a Server is Added or Removed**: Adjusting the auto-increment settings when a new server is added or an existing one is removed can be challenging. This might require a reconfiguration of the auto-increment steps on all servers to maintain the unique and gapless nature of ID generation.

2. **Universally unique identifier (UUID)**

What is UUID?

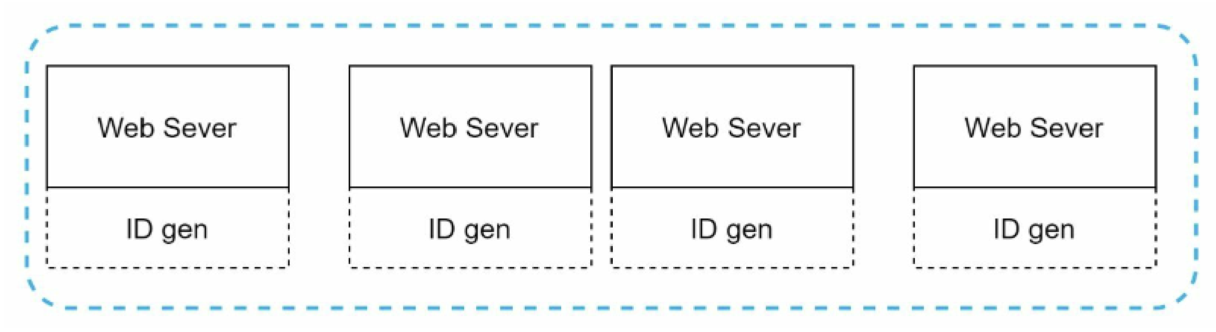
UUID is a 128-bit number used to identify information in computer systems.

A standard UUID is displayed in a 36-character string, with 32 hexadecimal digits divided by hyphens into five groups, represented as 8-4-4-4-12. This format includes hexadecimal digits (0-9 and A-F), which are each 4 bits in binary terms, thus totaling 128 bits (32 hex digits × 4 bits).

UUID has a very low probability of getting collusion. So, UUIDs can be generated independently without coordination between servers.

Example: `09c93e62-50b4-468d-bf8a-c07e1040bfb2`

How does it work?



Pros:

- Generating UUID is simple. No coordination between servers is needed so there will not
  be any synchronization issues.
- The system is easy to scale because each web server is responsible for generating IDs
  they consume. ID generator can easily scale with web servers.

Cons:

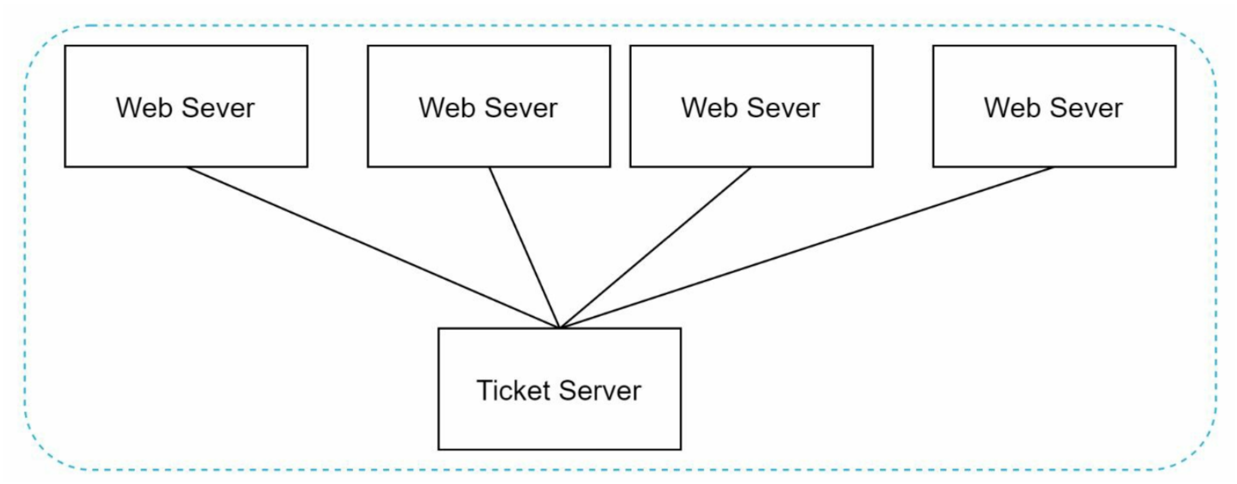
- IDs are 128 bits long, but our requirement is 64 bits.
- IDs do not go up with time.
- IDs could be non-numeric.

3. **Ticket Server**

What is a Ticket Server?

A "ticket server" in the context of implementing a unique ID generator in a distributed system refers to a centralized server responsible for issuing unique identifiers (or tickets) upon request. This approach is typically used in scenarios where ensuring the uniqueness of each ID across the entire system is critical.

How does it work?



- **Centralization**: The ticket server acts as a central authority that manages the creation and distribution of unique IDs. It maintains a counter or other mechanism to ensure that each ID it issues is unique.

- **ID Requests**: Whenever a component of the distributed system needs a new unique ID, it sends a request to the ticket server.
- **ID Issuance**: The ticket server increments its counter (or performs another operation to generate a unique ID) and sends the new ID back to the requester.
- **Synchronization**: The ticket server ensures that no two IDs are the same, even when there are multiple requests at the same time. This often involves locking mechanisms or atomic operations to handle concurrent accesses safely.

Pros:

- **Guaranteed Uniqueness**: Since all ID assignments are centralized, the ticket server can guarantee that each ID is unique across the entire system.
- **Simplicity**: The implementation and logic for ID generation are centralized, simplifying the management and generation process.

Cons:

- **Single Point of Failure**: The ticket server can become a bottleneck for the distributed system. If the ticket server goes down, no new IDs can be generated, and parts of the system may become non-functional.
- **Scalability Concerns**: As the demand for new IDs increases, the ticket server may struggle to keep up if not properly scaled. It also introduces latency since every ID generation requires communication with the central server.
- **Potential Performance Bottleneck**: The need to contact a central server every time an ID is needed can introduce latency, particularly in geographically distributed environments.

4. **Twitter snowflake approach**

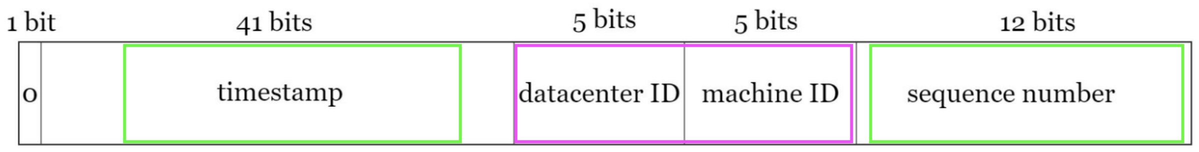


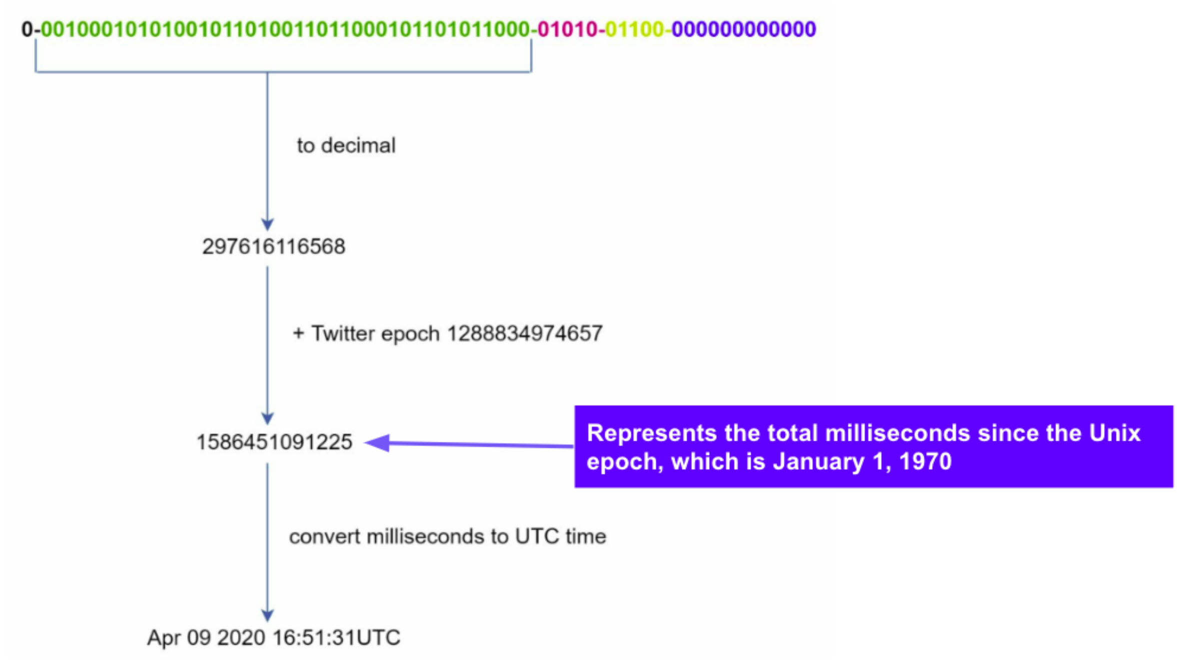
Figure 7-5

Each section is explained below.

- Sign bit: 1 bit. It will always be 0. This is reserved for future uses. It can potentially be used to distinguish between signed and unsigned numbers.
- Timestamp: 41 bits. Milliseconds since the epoch or custom epoch. We use Twitter snowflake default epoch 1288834974657, equivalent to Nov 04, 2010, 01:42:54 UTC.
- Datacenter ID: 5 bits, which gives us *2 ^ 5 = 32* datacenters.
- Machine ID: 5 bits, which gives us *2 ^ 5 = 32* machines per datacenter.
- Sequence number: 12 bits. For every ID generated on that machine/process, the sequence number is incremented by 1. The number is reset to 0 every millisecond.

Step 3 - Design deep dive

- Datacenter IDs and machine IDs:
  - chosen at the startup time, generally fixed once the system is up running.
  - Any changes in datacenter IDs and machine IDs require careful review since an accidental change in those values can lead to ID conflicts.
- Timestamp
  - generated when the ID generator is running.
  - 41 bits(2^41 - 1: 69 years)
  - grow with time
  -


0-00100010101001011010011011000101101011000-01010-01100-000000000000
to decimal
297616116568
+ Twitter epoch 1288834974657
1586451091225
Represents the total milliseconds since the Unix epoch, which is January 1, 1970
convert milliseconds to UTC time
Apr 09 2020 16:51:31UTC

- sequence numbers:
  - generated when the ID generator is running.
  - 12 bits(2^12 =4096 combinations)
  - for every ID generated on a machine/process, the sequence number is incremented by 1, and the number is reset to 0 every millisecond.

Step 4 - Wrap up

• Clock synchronization. In our design, we assume ID generation servers have the same clock. This assumption might not be true when a server is running on multiple cores. The same challenge exists in multi-machine scenarios. Solutions to clock synchronization are out of the scope of this book; however, it is important to understand the problem exists. Network Time Protocol is the most popular solution to this problem. For interested readers, refer to the reference material [4].

• Section length tuning. For example, fewer sequence numbers but more timestamp bits are effective for low concurrency and long-term applications.

• High availability. Since an ID generator is a mission-critical system, it must be highly available.