Design of a Unique ID Generator

Learn how to design a system that generates a unique ID.

We'll cover the following

In the previous lesson, we saw that we need unique identifiers for many use cases, such as identifying objects (for example, Tweets, uploaded videos, and so on) and tracing the execution flow in a complex web of services. Now, we'll formalize the requirements for a unique identifier and discuss three progressively improving designs to meet our requirements.

Requirements for unique identifiers

The requirements for our system are as follows:

- **Uniqueness**: We need to assign unique identifiers to different events for identification purposes.

- **Scalability**: The ID generation system should generate at least a billion unique IDs per day.
- **Availability**: Since multiple events happen even at the level of nanoseconds, our system should generate IDs for all the events that occur.
- **64-bit numeric ID**: We restrict the length to 64 bits because this bit size is enough for many years in the future. Let's calculate the number of years after which our ID range will wrap around.

  Total numbers available = 2^{64} = 1.8446744 x 10^{19}

  Estimated number of events per day = 1 billion = 10^9

  Number of events per year = 365 billion = 365×10^9

  Number of years to deplete identifier range = $\frac{2^{64}}{365 \times 10^9}$ = 50,539,024.8595 years

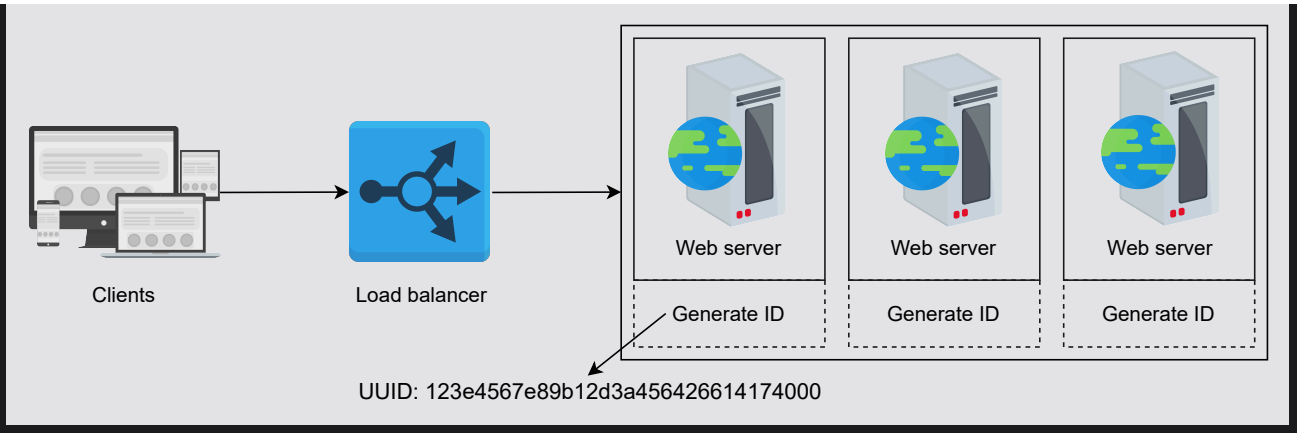
  64 bits should be enough for a unique ID length considering these calculations.

Let's dive into the possible solutions for the problem mentioned above.

First solution: UUID

A straw man solution for our design uses **UUIDs (universally unique IDs)**. This is a 128-bit number and it looks like $123e4567e89b12d3a456426614174000$ in hexadecimal. It gives us about 10^{38} numbers. UUIDs have different versions. We opt for version 4, which generates a pseudorandom number.

Each server can generate its own ID and assign the ID to its respective event. No coordination is needed for UUID since it's independent of the server. Scaling up and down is easy with UUID, and this system is also highly available. Furthermore, it has a low probability of collisions. The design for this approach is given below:



Generating a unique ID using the UUID approach

Point to Ponder

Q **(Select all that apply.)** What are the pros of using the UUID approach?

- [ ] **A)** It doesn't require synchronization between servers.
- [ ] **B)** It's a simple approach.
- [ ] **C)** It's scalable.
- [ ] **D)** It's available.

Submit Answer

Reset Quiz

Cons

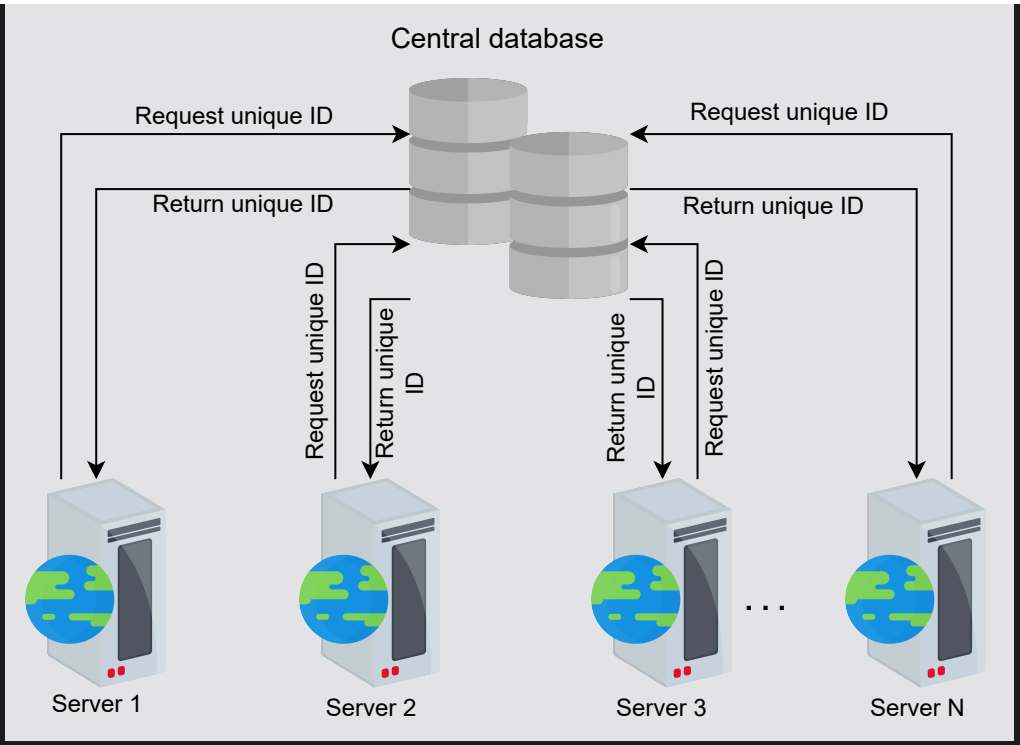
Using 128-bit numbers as primary keys makes the primary-key indexing slower, which results in slow inserts. A workaround might be to interpret an ID as a hex string instead of a number. However, non-numeric identifiers might not be suitable for many use cases. The ID isn't of 64-bit size. Moreover, there's a chance of duplication. Although this chance is minimal, we can't claim UUID to be deterministically unique. Additionally, UUIDs given to clients over time might not be monotonically increasing. The following table summarizes the requirements we have fulfilled using UUID:

Requirements Filled with UUID

	Unique	Scalable	Available	64-bit numeric ID
Using UUID	✗	✓	✓	✗

Second solution: using a database

Let's try mimicking the auto-increment feature of a database. Consider a central database that provides a current ID and then increments the value by one. We can use the current ID as a unique identifier for our events.



Using a central database to generate unique IDs

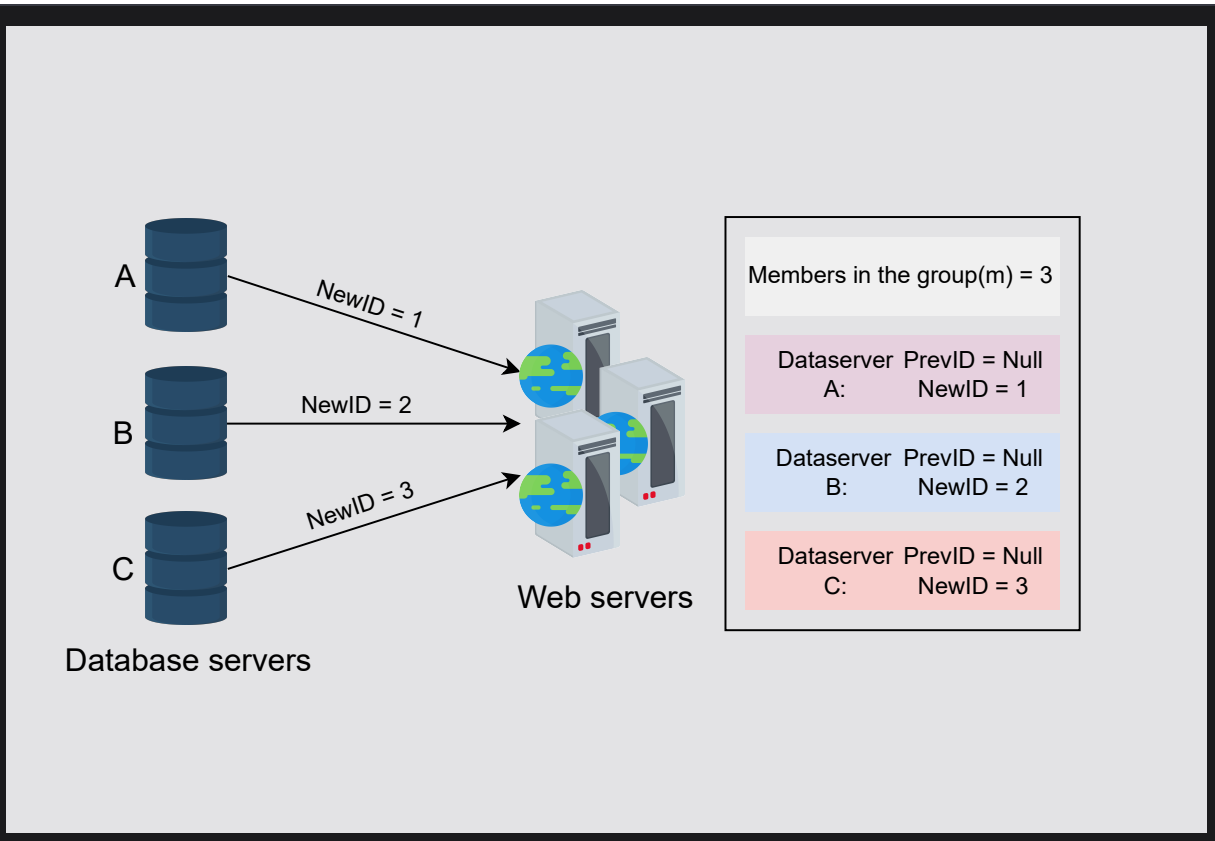
Point to Ponder

Question

What can be a potential problem of using a central database?

Show Answer

To cater to the problem of a single point of failure, we modify the conventional auto-increment feature that increments by one. Instead of incrementing by one, let's rely on a value `m`, where `m` equals the number of database servers we hav Each server generates an ID, and the following ID adds `m` to the previous value. This method is scalable and prevents the duplication of IDs. The following ima provides a visualization of how a unique ID is generated using a database:



Generating IDs using the value of m

1 of 2

Pros

This approach is scalable. We can add more servers, and the value of `m` will be updated accordingly.

Cons

Though this method is somewhat scalable, it's difficult to scale for multiple data centers. The task of adding and removing a server can result in duplicate IDs. For example, suppose `m=3`, and server A generates the unique IDs 1, 4, and 7. Server B generates the IDs 2, 5, and 8, while server C generates the IDs 3, 6, and 9. Server B faces downtime due to some failure. Now, the value `m` is updated to 2. Server A generates 9 as its following unique ID, but this ID has already been generated by server C. Therefore, the IDs aren't unique anymore.

The table below highlights the limitations of our solution. A unique ID generation system shouldn't be a **single point of failure (SPOF)**. It should be scalable and available.

Requirements Filled by UUID versus Using a Database

	Unique	Scalable	Available	64-bit numeric ID
Using UUID	✗	✓	✓	✗
Using a database	✗	✓	✓	✓

Third solution: using a range handler

Let's try to overcome the problems identified in the previous methods. We can use ranges in a central server. Suppose we have multiple ranges for one to two billion, such as 1 to 1,000,000; 1,000,001 to 2,000,000; and so on. In such a case, a central microservice can provide a range to a server upon request.

Any server can claim a range when it needs it for the first time or if it runs out of the range. Suppose a server has a range, and now it keeps the start of the range in a local variable. Whenever a request for an ID is made, it provides the local variable value to the requestor and increments the value by one.

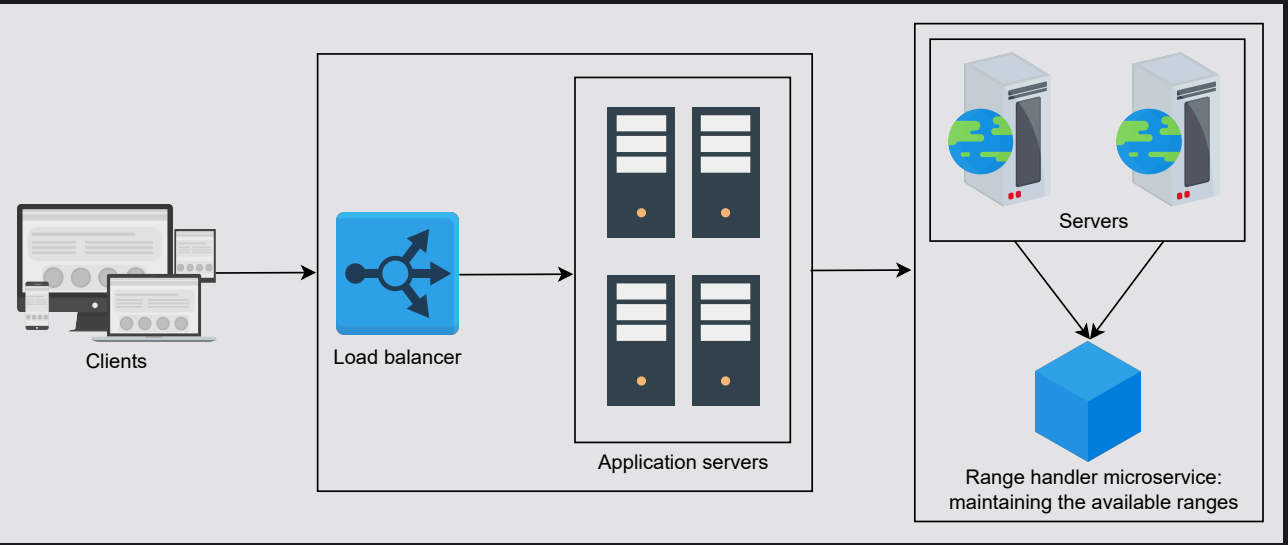
Let's say server 1 claims the number range 300,001 to 400,000. After this range claim, the user ID 300,001 is assigned to the first request. The server then returns 300,002 to the next user, incrementing its current position within the range. This continues until user ID 400,000 is released by the server. The application server then queries the central server for the next available range and repeats this process.

This resolves the problem of the duplication of user IDs. Each application serv can respond to requests concurrently. We can add a load balancer over a set of

servers to mitigate the load of requests.

We use a microservice called **range handler** that keeps a record of all the taken and available ranges. The status of each range can determine if a range is available or not. The state—that is, which server has what range assigned to it—can be saved on a replicated storage.

This microservice can become a single point of failure, but a **failover server** acts as the savior in that case. The failover server hands out ranges when the main server is down. We can recover the state of available and unavailable ranges from the latest checkpoint of the replicated store.



Design of the range handler microservice

Imagine a scenario where you're planning to bring a new data center online to accommodate increasing user requests. How would you modify your range allocation strategy to accommodate this change?

Want to know the correct answer?

Adjusting range allocation for new data center

H1 H2 H3 | B I | | | | | | |

Provide your answer here!

Pros

This system is scalable, available, and yields user IDs that have no duplicates. Moreover, we can maintain this range in 64 bits, which is numeric.

Cons

We lose a significant range when a server dies and can only provide a new range once it’s live again. We can overcome this shortcoming by allocating shorter ranges to the servers, although ranges should be large enough to serve identifiers for a while.

The following table sums up what this approach fulfills for us: