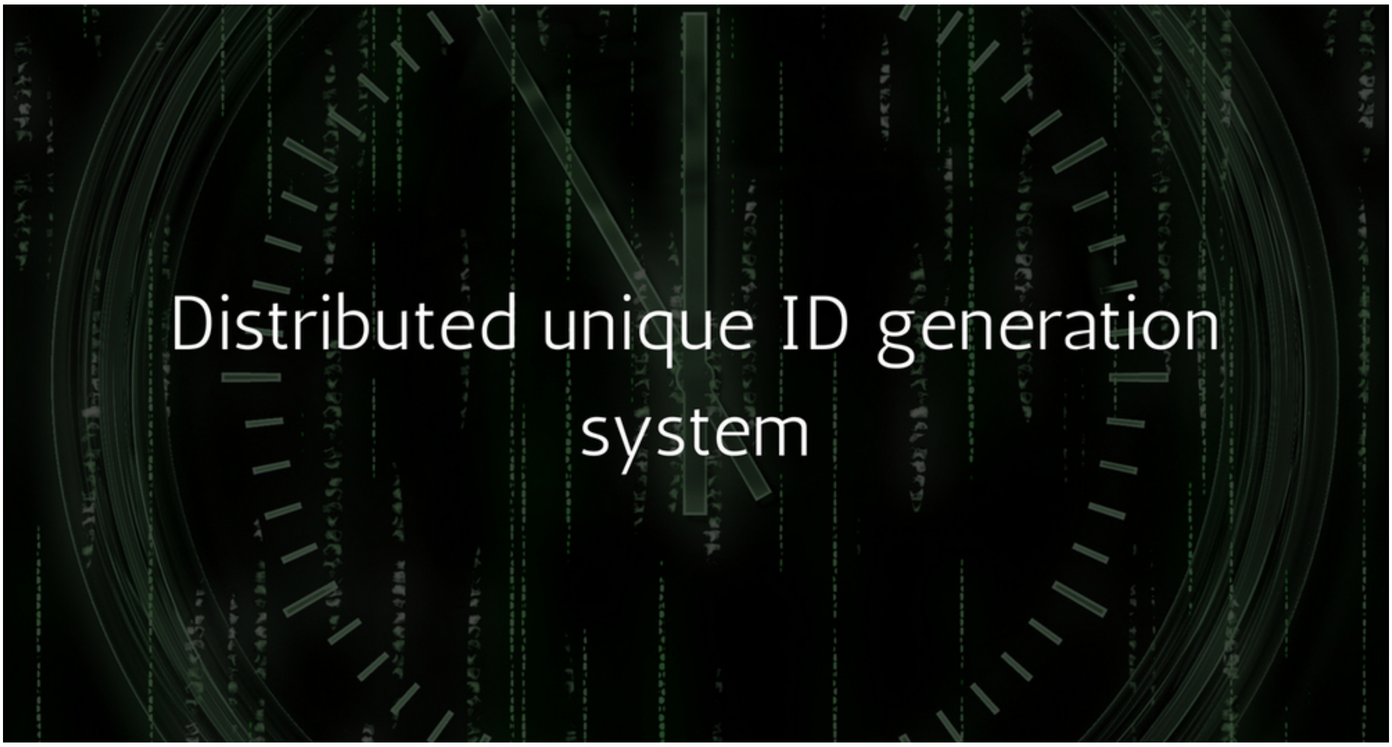
CALLICODER

Generating unique IDs in a distributed environment at high scale.

Rajeev Singh • System Design • Jun 8, 2018 • 6 mins read



I was recently working on an application that had a sharded MySQL database. While working on this application, I needed to generate unique IDs that could be used as the primary keys in the tables.

When you're working with a single MySQL database, you can simply use an auto-increment ID as the primary key, But this won't work in a sharded MySQL database.

So I looked at various existing solutions for this, and finally wrote a simple 64-bit unique ID generator that was inspired by a similar service by Twitter called Twitter snowflake.

In this article, I'll share a simplified version of the unique ID generator that will work for any use-case of generating unique IDs in a distributed environment, not just sharded databases.

I'll also outline other existing solutions and discuss their pros and cons.

CALLICODER

Existing Solutions

UUID

UUIDs are 128-bit hexadecimal numbers that are globally unique. The chances of the same UUID getting generated twice is negligible.

The problem with UUIDs is that they are very big in size and don't index well. When your dataset increases, the index size increases as well and the query performance takes a hit.

MongoDB's ObjectId

MongoDB's ObjectIDs are 12-byte (96-bit) hexadecimal numbers that are made up of -

- a 4-byte epoch timestamp in seconds,
- a 3-byte machine identifier,
- a 2-byte process id, and
- a 3-byte counter, starting with a random value.

This is smaller than the earlier 128-bit UUID. But again the size is relatively longer than what we normally have in a single MySQL auto-increment field (a 64-bit bigint value).

Database Ticket Servers

This approach uses a centralized database server to generate unique incrementing IDs. It's like a centralized auto-increment. This approach is used by Flickr.

The problem with this approach is that the ticket server can become a write bottleneck. Moreover, you introduce one more component in your infrastructure that you need to manage and scale.

Twitter Snowflake

Twitter snowflake is a dedicated network service for generating 64-bit unique IDs at high scale. The IDs generated by this service are roughly time sortable.

The IDs are made up of the following components:

- Epoch timestamp in millisecond precision - 41 bits (gives us 69 years with a custom epoch)
- Configured machine id - 10 bits (gives us up to 1024 machines)
- Sequence number - 12 bits (A local counter per machine that rolls over every 4096)

The IDs generated by twitter snowflake fits in 64-bits and are time sortable, which is great. That's what we want.

But If we use Twitter snowflake, we'll again be introducing another component in our infrastructure that we need to maintain.

Distributed 64-bit unique ID generator inspired by Twitter Snowflake

Finally, I wrote a simple sequence generator that generates 64-bit IDs based on the concepts outlined in the Twitter snowflake service.

The IDs generated by this sequence generator are composed of -

- **Epoch timestamp in milliseconds precision** - 41 bits. The maximum timestamp that can be represented using 41 bits is 2^{41} - 1, or 2199023255551, which comes out to be `Wednesday, September 7, 2039 3:47:35.551 PM`. That gives us 69 years with respect to a custom epoch.
- **Node ID** - 10 bits. This gives us 1024 nodes/machines.
- **Local counter per machine** - 12 bits. The counter's max value would be 4095.

The remaining 1-bit is the signed bit and it is always set to 0.

Your microservices can use this Sequence Generator to generate IDs independently. This is efficient and fits in the size of a bigint.

Here is the complete program -

```java
import java.net.NetworkInterface;
import java.security.SecureRandom;
import java.time.Instant;
import java.util.Enumeration;

/**
 * Distributed Sequence Generator.
 * Inspired by Twitter snowflake: https://github.com/twitter/snowflake/tree/snowflake-2010
 *
 * This class should be used as a Singleton.
```

CALLICODER

```java
public class SequenceGenerator {
    private static final int UNUSED_BITS = 1; // Sign bit, Unused (always set to 0)
    private static final int EPOCH_BITS = 41;
    private static final int NODE_ID_BITS = 10;
    private static final int SEQUENCE_BITS = 12;

    private static final int maxNodeId = (int)(Math.pow(2, NODE_ID_BITS) - 1);
    private static final int maxSequence = (int)(Math.pow(2, SEQUENCE_BITS) - 1);

    // Custom Epoch (January 1, 2015 Midnight UTC = 2015-01-01T00:00:00Z)
    private static final long CUSTOM_EPOCH = 1420070400000L;

    private final int nodeId;

    private volatile long lastTimestamp = -1L;
    private volatile long sequence = 0L;

    // Create SequenceGenerator with a nodeId
    public SequenceGenerator(int nodeId) {
        if(nodeId < 0 || nodeId > maxNodeId) {
            throw new IllegalArgumentException(String.format("NodeId must be between %d and %d", 0, maxNodeId)
        }
        this.nodeId = nodeId;
    }

    // Let SequenceGenerator generate a nodeId
    public SequenceGenerator() {
        this.nodeId = createNodeId();
    }

    public synchronized long nextId() {
        long currentTimestamp = timestamp();

        if(currentTimestamp < lastTimestamp) {
            throw new IllegalStateException("Invalid System Clock!");
        }

        if (currentTimestamp == lastTimestamp) {
            sequence = (sequence + 1) & maxSequence;
            if(sequence == 0) {
                // Sequence Exhausted, wait till next millisecond.
                currentTimestamp = waitNextMillis(currentTimestamp);
            }
        } else {
            // reset sequence to start with zero for the next millisecond
            sequence = 0;
        }
```

CALLICODER

```java
        long id = currentTimestamp << (NODE_ID_BITS + SEQUENCE_BITS);
        id |= (nodeId << SEQUENCE_BITS);
        id |= sequence;
        return id;
    }


    // Get current timestamp in milliseconds, adjust for the custom epoch.
    private static long timestamp() {
        return Instant.now().toEpochMilli() - CUSTOM_EPOCH;
    }

    // Block and wait till next millisecond
    private long waitNextMillis(long currentTimestamp) {
        while (currentTimestamp == lastTimestamp) {
            currentTimestamp = timestamp();
        }
        return currentTimestamp;
    }

    private int createNodeId() {
        int nodeId;
        try {
            StringBuilder sb = new StringBuilder();
            Enumeration<NetworkInterface> networkInterfaces = NetworkInterface.getNetworkInterfaces();
            while (networkInterfaces.hasMoreElements()) {
                NetworkInterface networkInterface = networkInterfaces.nextElement();
                byte[] mac = networkInterface.getHardwareAddress();
                if (mac != null) {
                    for(int i = 0; i < mac.length; i++) {
                        sb.append(String.format("%02X", mac[i]));
                    }
                }
            }
            nodeId = sb.toString().hashCode();
        } catch (Exception ex) {
            nodeId = (new SecureRandom().nextInt());
        }
        nodeId = nodeId & maxNodeId;
        return nodeId;
    }
}
```

The above generator uses the system’s MAC address to create a unique identifier for the Node. You can also supply a NodeID to the sequence generator. That will guarantee uniqueness.

Let's now understand how it works. Let's say it's `June 9, 2018 10:00:00 AM GMT`. The epoch timestamp for this particular time is `1528538400000`.

First of all, we adjust our timestamp with respect to the custom epoch-

```
currentTimestamp = 1528538400000 - 1420070400000 // 108468000000 (Adjust for custom epoch)
```

Now, the first 41 bits of the ID (after the signed bit) will be filled with the epoch timestamp. Let's do that using a left-shift -

```
id = currentTimestamp << (10 + 12)
```

Next, we take the configured node ID and fill the next 10 bits with the node ID. Let's say that the `nodeId` is 786 -

```
id |= nodeId << 12
```

Finally, we fill the last 12 bits with the local counter. Considering the counter's next value is 3450, i.e. `sequence = 3450`, the final ID is obtained like so -

```
id |= sequence  // 454947766275219456
```

That gives us our final ID.

More Learning Resources

- UUID (Universally unique identifier)
- MongoDB ObjectId
- Ticket Servers: Distributed Unique Primary Keys on the Cheap
- Twitter Snowflake
- Sharding & IDs at Instagram
- Distributed sequence number generation?

CALLICODER

Liked the Article? Share it on Social media!

Twitter Facebook Linkedin Reddit

Deploying / Hosting Spring Boot...

callicoder.com

Dockerizing your Spring Boot applications

callicoder.com

Spring Boot + Spring Security + JWT +...

callicoder.com

Kotlin Infix Notation - Make function calls...

callicoder.com

Spring Boot + Spring Security + JWT +...

callicoder.com

Building Docker Containers for Go...

callicoder.com

Spring Boot Fil Upload / Downl

callicoder.com

Comments **Community** **Privacy Policy** Ava

Recommend 7 Tweet Share **Sort by Newest**

Join the discussion…

Imran Bijapuri • 6 months ago

How is this any different from the Database Ticket Servers ? You are still dependent on a single point.
I am just trying to understand how the service works in case of increase in the number of writes. Won't it behave like a bottleneck just like Database Ticket Servers ?

Reply • Share ›

CALLICODER

NODE_ID_BITS) - 1);
private static final int maxSequence = (int)(Math.pow(2, SEQUENCE_BITS) - 1);

the -1 should be removed as the last number in the sequence will never be used otherwise

^ | ˅ • Reply • Share ›

cheng dong • 8 months ago

you use 10bit to store nodeId, it's reasonable a big chance for two machine have one nodeId, and they are very possible to generate sequence from 0 - little numbers. so if you got two machine with same mac hash, it is for sure they will produce same id.

^ | ˅ • Reply • Share ›

Nghĩa • 8 months ago • edited

If i want a 8 digit length id, how can i do this?

^ | ˅ • Reply • Share ›

kgoya • 10 months ago

Thanks for such a nice post. I am getting ids generated which are jumping from "58309118368166428" to "61327561782755120" in a month duration. I am wondering whether it will last for 139 years as suggested.

^ | ˅ • Reply • Share ›

Bow Archer • a year ago • edited

Did you consider the `DelayQueue` mechanism instead of simply blocking on a while loop for exhaustion check? I've not done comparisons on this code, but theoretically, on a single core machine, this will block the entire process, and at best case probably use more cycles on a faster machine.

Optimization to the "nextId" method would be to:

```
// inline tends to be faster than using |= and reassigning
the stack variable
return (currTs << (BITS_TOTAL - BITS_EPOC)) | (nodeId <<
(BITS_TOTAL - BITS_EPOC - BITS_NODE)) | seq;
```

One other optimization would be to encode the mac address with this loop:

```
private static final String MAC_CODE = "%02X";
for (byte macPart:mac) {
    sb.append(String.format(MAC_CODE, macPart));
}
```

which will avoid the extra stack variable, length check, and increment operations as you do the work. Also hoist the string to a constant.

^ | ˅ • Reply • Share ›

Lokesh Niranjan • a year ago

Hi rajeev can you brief with bullet points. which explains the

CALLICODER

Deyan Gyurdzhekliev • a year ago

Hi, Rajeev ! Good Article. What does adjusted timestamp give us as an advantage? Why just don't use Instant.now().toEpochMilli() as timestamp value?

1 • Reply • Share ›

Dani • a year ago

Good article. Two minor problems though. It's easy for the nextId function to throw since it gets the current time out of the synchronized block. Also, the sequence and lastTimestamp are not thread-safe. You can mark them as volatile.

• Reply • Share ›

Rajeev Singh Mod → Dani • a year ago

Thanks Dani for raising this. I think moving the `synchronized` keyword to method is the best fix here. I have updated the post

3 • Reply • Share ›

Lokesh Niranjan → Rajeev Singh • a year ago

Hi rajeev thanks for the article

• Reply • Share ›

Dani → Rajeev Singh • a year ago

Yep, but notice that the variables sequence and lastTimestamp are still not thread-safe. Marking them as volatile should do the trick.

• Reply • Share ›

Andrew → Dani • a year ago

Actually, I can't see why volatile matters. Both sequence and lastTimestamp are only ever accessed by a thread which must hold the instance monitor lock by virtue of the nextId() method being synchronized. All reads and writes on that single thread must be visible to that thread.

CALLICODER

CalliCoder

Software Development Tutorials written from the heart!

RESOURCES

About & Contact

Advertise

Recommended Books

Recommended Courses

Privacy Policy

Sitemap

TOOLS

URL Encoder

URL Decoder

Base64 Encoder

Base64 Decoder

QRCodeBit

ASCII Table

JSON Formatter

CONNECT

Twitter

Github

Facebook

Linkedin

Reddit