Snowflake ID

Finn Behrend

April 2, 2023

Complete sheet and specification about Twitter's Snowflake concept

Table of Contents

1 Introduction

The 'Snowflake' concept was originally invented by Twitter and is designed to generate short and guaranteed unique ID[1] numbers at high scale.

1.1 Motivation

Twitter invented the Snowflake concept because they were moving away from MySQL (Database) and moved to Cassandra because of scaling reasons, so they needed a new way to generate unique ID's based on the fact that Cassandra does and should not provide a sequential id generation facility.

2 Capacity

Snowflakes are designed to withstand the following specifications with ease:

- 10k ID generations per second per process
- Response rate of 2ms

In large data centers, machines generating ID's should not have to coordinate with each other. That's what makes Snowflakes Concept so brilliant as computers do not have to communicate with each other during or after the id creation process.

3 Scheme

Example of a Snowflake ID from Discord[2] we'll use throughout the sheet: `555780371086704681`

A Snowflake ID consists of a 64-bit number (e.g. a `uint64`), depending on implementation they mostly are handled as strings to prevent integer overflows in some languages.

Being a 64-bit integer makes the Snowflake ID's practical maximum 2^{64} or $(\text{FFFFFFFFFFFFFFFF})_{16}$.

4 Binary Breakdown

A Snowflake ID is constructed by chaining binary data and then converting to decimal.

`0000011110110110100001110100110110000011111 00001 00000 000000101001`

Figure 1: Binary Breakdown of the example ID

Field	Bits	Description
Timestamp	42	Millisenconds since the standard UNIX Epoch
Internal worker ID	5	Usually incremented ID of the worker
Internal process ID	5	Usually incremented ID of the process in the worker
Increment / Sequence	12	For every ID generated on process, this number is incremented

[1] Short for **Identifier**, is a name that identifies (that is, labels the identity of) either a unique object or a unique class of objects, where the 'object' or class may be an idea, physical countable object (or class thereof), or physical non-countable substance (or class thereof).

[2] Discord is a VoIP and instant messaging social platform. Users have the ability to communicate with voice calls, video calls, text messaging, media and files in private chats or as part of communities called 'servers'.

5 Retrieving data from a Snowflake ID

5.1 Timestamp Field

The timestamp field is the first field and has `42` bits in total. It contains the UNIX Timestamp[3] when the Snowflake ID was generated. Implementations like to use custom epochs to lower the timestamp at the beginning of use, and postpone the 'Year-3038' problem. Usually the first day of January in a Year is taken.

```
long snowflake = 555780371086704681;
long customEpoch = 1420070400000;
// As corresponding example for the Snowflake ID,
// Discord's custom Epoch is used

long timestamp = (snowflake >> 22) + customEpoch;
```

Figure 2: Example of timestamp retrieval in C with custom epoch

```
long snowflake = 555780371086704681;
long timestamp = (snowflake >> 22);
```

Figure 3: Example of timestamp retrieval in C without custom epoch

5.2 Internal worker-/process ID Fields

The 'Internal worker ID' and 'Internal process ID' fields are the second and third fields and both hold 5 bits of data.

These fields hold simply what their names tell us, the former field holds the unique ID of the worker machine that created the Snowflake, and the second field holds the unique process ID living in that worker machine, both usually incremented.

```
long snowflake = 555780371086704681;
int workerId = (snowflake & 0x3E0000) >> 17;
```

Figure 4: Example of worker ID retrieval in C

```
long snowflake = 555780371086704681;
int processId = (snowflake & 0x1F000) >> 12;
```

Figure 5: Example of process ID retrieval in C

[3]Unix time is a date and time representation widely used in computing. It measures time by the number of seconds or milliseconds that have elapsed since `00:00:00 UTC on 1 January 1970`, the beginning of the Unix epoch, less adjustments made due to leap seconds.

5.3 Increment Field

The increment field, with consisting of 12 bits is the last chained data in the Snowflake ID. It is incremented for every ID that is generated on that process.

```
long snowflake = 555780371086704681;
int increment = snowflake & 0xFFF;
```

Figure 6: Example of increment retrieval in C

5.4 Complete Implementation Example

This is what a proper code retrieving all fields from a Snowflake ID may look like:

```
long snowflake = 555780371086704681;
long customEpoch = 1420070400000;

long timestamp = (snowflake >> 22) + customEpoch;
int workerId = (snowflake & 0x3E0000) >> 17;
int processId = (snowflake & 0x1F000) >> 12;
int increment = snowflake & 0xFFF;
```

Figure 7: Example of proper retrieval implementation with custom epoch in C

```
long snowflake = 555780371086704681;

long timestamp = (snowflake >> 22);
int workerId = (snowflake & 0x3E0000) >> 17;
int processId = (snowflake & 0x1F000) >> 12;
int increment = snowflake & 0xFFF;
```

Figure 8: Example of proper retrieval implementation without custom epoch in C

6 Generating Snowflake ID's

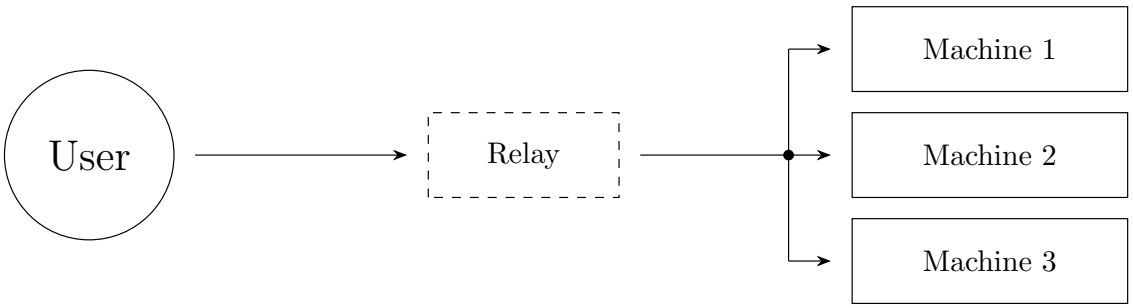

Figure 9: Example Infrastructure of a ID generating network

Assuming we have a infrastructure as shown above, and a user is requesting a registration that gets sent to a relay, which then relays the request of one of our 3 machines (which generates a ID for that user), and let our ID generating function look like this:

```
        long snowflake(int workerId, int processId) {
                long timestamp = unix_time();
                static int increment = -1;
                increment++;

                return (timestamp << 22)
                | (workerId << 17)
                | (processId << 12)
                | increment;
            }
```

Figure 10: Basic Example of a snowflake generating function in C

As you can see, we have two arguments that can be passed into that function, a `workerId` variable and a `processId` variable. Every worker machine should have a different worker ID, while every process or thread of that worker should have a different process ID (both usually incremented).

First, we are getting the current UNIX Timestamp. It is highly recommended to use NTP[4] to keep your system clock accurate.

Now we are defining a static integer to make a consistent incremental for each call of the function, and increase it by 1 after initialisation.

At last, we are shifting the data into the right positions and return the final Snowflake ID.

[4]The Network Time Protocol (short **NTP**) is a networking protocol for clock synchronization between computer systems over packet-switched, variable-latency data networks.

6.1 Practical Test

I've let the first machine generate three snowflakes, two immediately after each other and one delayed 2 seconds after the first two ID's.

```
704692358047032934 4
704692358047452364 9
704692358886313165 0
```

Figure 11: The three Snowflake ID's that were generated

Let's break them down.

```
11000011100101110110011101010010000010101 00001 00000 000000000000
```

Figure 12: Binary Breakdown of the first ID

```
11000011100101110110011101010010000010110 00001 00000 000000000001
```

Figure 13: Binary Breakdown of the second ID

```
11000011100101110110011101010101111100110 00001 00000 000000000010
```

Figure 14: Binary Breakdown of the third ID

As you can see, all ID's were generated on worker one and process zero (red and green fields).

Also, the increment field properly incremented from one, to two, to three.

Now, let's look at the timestamps (non-custom epoch):

```
1680117507093
1680117507094
1680117509094
```

Figure 15: The timestamps of the three Snowflake ID's that were generated

As you can see, on the first two generations the timestamp only changed by `1 ms` which is pure machine latency. But looking at the difference from the second and third timestamp we see a `2000 ms` difference, which is exactly what we wanted!

For a custom epoch, just subtract your epoch from the UNIX epoch and then compile to a Snowflake.

7 2038 Problem

The 2038 problem, also known as the Unix Y2K bug, refers to a limitation of the Unix timestamp, which is used to represent time as a 32-bit signed integer, and is set to overflow on `January 19, 2038`. This means that any software relying on the Unix timestamp to represent dates beyond that point will encounter errors or fail completely.

As we already know, a Snowflake is a 64-bit integer and a part of it consists of a constantly growing UNIX Timestamp. Since the Snowflake is in fact able to hold a 42-bits instead of just 32, the problem gets put off to `May 15, 2109` but, is still present.

One approach, is to use custom epochs which postpone the problem for a time. A good choice for a custom epoch is a time before you started utilising Snowflake ID's, since Snowflake ID's usually never require to have a timestamp that goes back before the implementation time.
Of course, this is not the case if you already have and plan to migrate from an old ID System, and want to convert ID's.

After all, those approaches only delay the problem, so its always good to have it back in mind.

8 Constraints

9 References