Distributed Unique ID Generation System

Requirements

- Design a monotonically increasing unique id generation system.

Approach 1 : Timestamp

- Time is unique and is monotonically increasing.

```
id_t UniqueID() {
  struct timeval tv;
  gettimeofday(&tv, NULL);
  return cat(tv.tv_sec,
             tv.tv_usec);
}
```

- Cons: You might call the function at the same time from two different computers.

Approach 2: Timestamp + MachineId

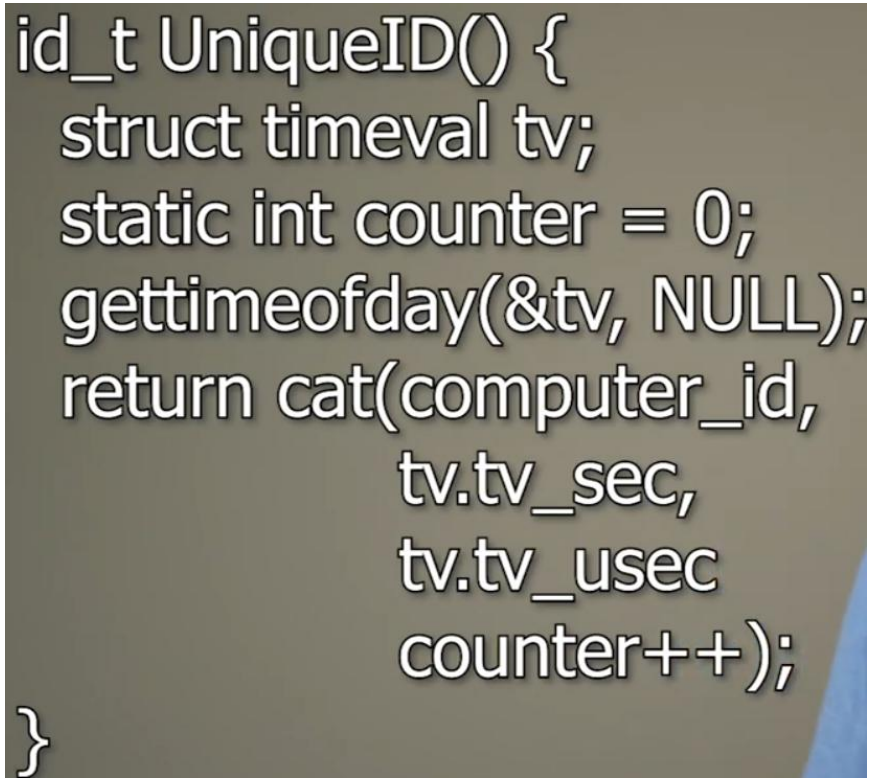
Identity:

- IP address
- Ethernet HW address
- CPU serial number

```
id_t UniqueID() {
  struct timeval tv;
  gettimeofday(&tv, NULL);
  return cat(computer_id,
             tv.tv_sec,
             tv.tv_usec);
}
```

Cons: What if you invoke the function twice in the same microsecond. We need some tie breaker.

Approach 3: Timestamp + MachineId + Counter



```
id_t UniqueID() {
  struct timeval tv;
  static int counter = 0;
  gettimeofday(&tv, NULL);
  return cat(computer_id,
             tv.tv_sec,
             tv.tv_usec
             counter++);
}
```

Available Solutions

1. 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.

2. Database Ticket Server

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.

This technique database support as we want INSERT ON DUPLICATE KEY UPDATE feature. It means, if we try to query the db on the any column, if the data exists then update the data else insert it.

MySQL achieves it using **REPLACE**

REPLACE works exactly like INSERT, except that if an old row in the table has the same value as a new row for a PRIMARY KEY or a UNIQUE index, the old row is deleted before the new row is inserted.

We want to take advantage of this delete if exists feature. If we have a auto increment id column, if it is deleted then it will be inserted with incremented id.

- A Flickr ticket server is a dedicated database server, with a single database on it, and in that database there are tables like Tickets32 for 32-bit IDs, and Tickets64 for 64-bit IDs.
- The Tickets64 schema looks like:

```
CREATE TABLE `Tickets64` (
  `id` bigint(20) unsigned NOT NULL auto_increment,
  `stub` char(1) NOT NULL default '',
  PRIMARY KEY  (`id`),
  UNIQUE KEY `stub` (`stub`)
) ENGINE=MyISAM
```

SELECT * from Tickets64 returns a single row that looks something like:

```
+-------------------+------+
| id                | stub |
+-------------------+------+
| 72157623227190423 |    a |
+-------------------+------+
```

When I need a new globally unique 64-bit ID I issue the following SQL:

REPLACE INTO Tickets64 (stub) VALUES ('a');
SELECT LAST_INSERT_ID();

- How to avoid SPOF?

You really really don't know want provisioning your IDs to be a single point of failure. We achieve "high availability" by running two ticket servers. At this write/update volume replicating between the boxes would be problematic, and locking would kill the performance of the site. We divide responsibility between the two boxes by dividing the ID space down the middle, evens and odds, using:

```
TicketServer1:
auto-increment-increment = 2
auto-increment-offset = 1

TicketServer2:
auto-increment-increment = 2
auto-increment-offset = 2
```

We round robin between the two servers to load balance and deal with down time. The sides do drift a bit out of sync, I think we have a few hundred thousand more odd number objects then evenly numbered objects at the moment, but this hurts no one.

3. 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 extra 1 bit is reserved for future purposes. Since the IDs use timestamp as the first component, they are time sortable.

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.

Our Solution : 64 Bit ID Generator based on Snowflake

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 219902325555l, 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.