**Chapter 7**

Design A Unique ID Generator in Distributed Systems

**High Level Designs**

Multiple design options

* Multi-master replication
* Universally unique identifier (UUID)
* Ticket server
* Twitter snowflake approach

*Multi-master replication*

* Concept
  + Uses the databases’ “auto\_increment feature
  + Instead of one, increments by the number of database servers in use
* Cons
  + Hard to scale with multiple data centers
  + IDs do not go up with time across multiple servers
  + Does not scale well when servers are added or removed

*UUIDs* – Universal Unique Identifier

* Concept
  + a 128-bit number used to identify information in computer systems

Ex.) 09c93e62-50b4-468d-bf8a-c07e1040bfb2

* + Very low probability of getting collisions
  + Can be generated independently without coordination between servers
* Pros
  + No synchronization issues since servers generate independently
  + Easy to scale
* Cons
  + 128 bits long
  + IDs do not increment with time
  + IDs are non-numeric

*Ticket Server (Flickr’s Approach)*

* Developed by Flickr to generate primary keys in distributed setup
* Concept
  + Have a separate database used for only auto increment IDs across servers
* Flickr ticket server – a dedicated database server with a single database on it
  + Tables like Tickets32 for 32-bit IDs, and Ticket64 for 64-bit IDs
  + Sequences for photos, accounts, offline tasks, and groups, etc.
  + Runs 2 servers dividing ID space to evens and odds
* Implementation
  + Uses “INSERT ON DUPLICATE KEY UPDATE” statement in MySQL
* Pros
  + Numeric IDs
  + Index very quickly and optimized for MySQL
  + Easy to implement and works for small to medium-scale applications
  + Makes reporting and debugging more straightforward and enables caching
* Cons
  + Single point of failure if using single ticket server
  + Data synchronization issues if using multiple ticket servers

*Twitter snowflake approach*

* Design Requirements
  + Support tens of thousands of IDs per second in a highly available manner
  + IDs need to be roughly sortable – tweet A and B posted around the same time should be found near one another
  + Must fit into 64-bits
* Concept
  + Have a specific ID format, dividing into different section

|  |  |  |
| --- | --- | --- |
| **Section** | **Bits** | **Description** |
| Sign bit | 1 | Reserved for future uses. Can potentially be used to distinguish between signed or unsigned numbers |
| Timestamp | 41 | Milliseconds since the epoch or custom epoch. Twitter Snowflake default epoch 1288834974657 (equivalent to Nov 4, 2010, 1:42:54 UTC) |
| Datacenter ID | 5 | datacenters |
| Machine ID | 5 | machines per datacenter |
| Sequence number | 12 | For every ID generated on a particular machine/process, the sequence number increments by 1. Number is reset to zero every millisecond. Supporting 4096 entries per machine per millisecond |

*A close up of a label

Description automatically generated*

* Datacenter IDs and machine IDs are chosen at startup time
* Timestamps and sequence numbers are generated while running

*Timestamps*

* How to derive UTC time from the timestamp
  + Number of milliseconds is obtained from timestamp bits
  + Add the Twitter epoch number (1288834974657)
  + Convert milliseconds to UTC time
* Maximum timestamp that can be represented in 41 bits is 2199023255551
  + Roughly 69 years before the custom epoch needs to be renewed or a new technique needs to be adopted

**Additional Points**

* Clock synchronization – we assumed the servers have the same clock, which may not be true when a server is running on multiple cores
  + Network Time Protocol is most popular solution to this problem
* Section length tuning – fewer sequence numbers but more timestamp bits are effective for low concurrency and long-term applications