

You have **2** free member-only stories left this month. [Sign up for Medium and get an extra one](#)

# What Are Snowflake IDs?

How to generate unique IDs in a distributed environment at scale



Nassos Michas

[Follow](#)

Jun 22, 2020 · 5 min read ★



Photo by [Aaron Burden](#) on [Unsplash](#)

Generating unique identifiers is a task all programmers have had to deal with at some point in the course of an application's development lifecycle. Unique IDs allow us to

properly identify data objects, persist them, retrieve them, and have them participate in complex relational patterns.

But how are these unique IDs generated and which approach works best at various scales of load? How do IDs remain unique in a distributed environment where multiple computing nodes compete for the next available ID?

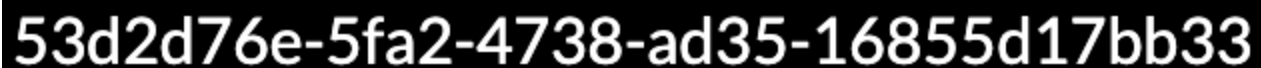
In this piece, I'll discuss three of the most common techniques that work — from a small, single-node scale to a Twitter-scale.

## Universal Unique Identifiers — UUIDs

Universal Unique Identifiers is a well-known concept that's been used in software for years. A UUID is a 128-bit number that, when generated in a controlled and standardised way, provides an extremely large keyspace, virtually eliminating the possibility of collision.

A UUID is a synthetic ID comprising of several distinct parts, such as time, the node's MAC address, or an MD5 hashed namespace. To accommodate all these combinations there have been multiple versions of the UUID specification over the years, notably versions 1 and 4. However, other versions may be of interest to you depending on your data and business domain.

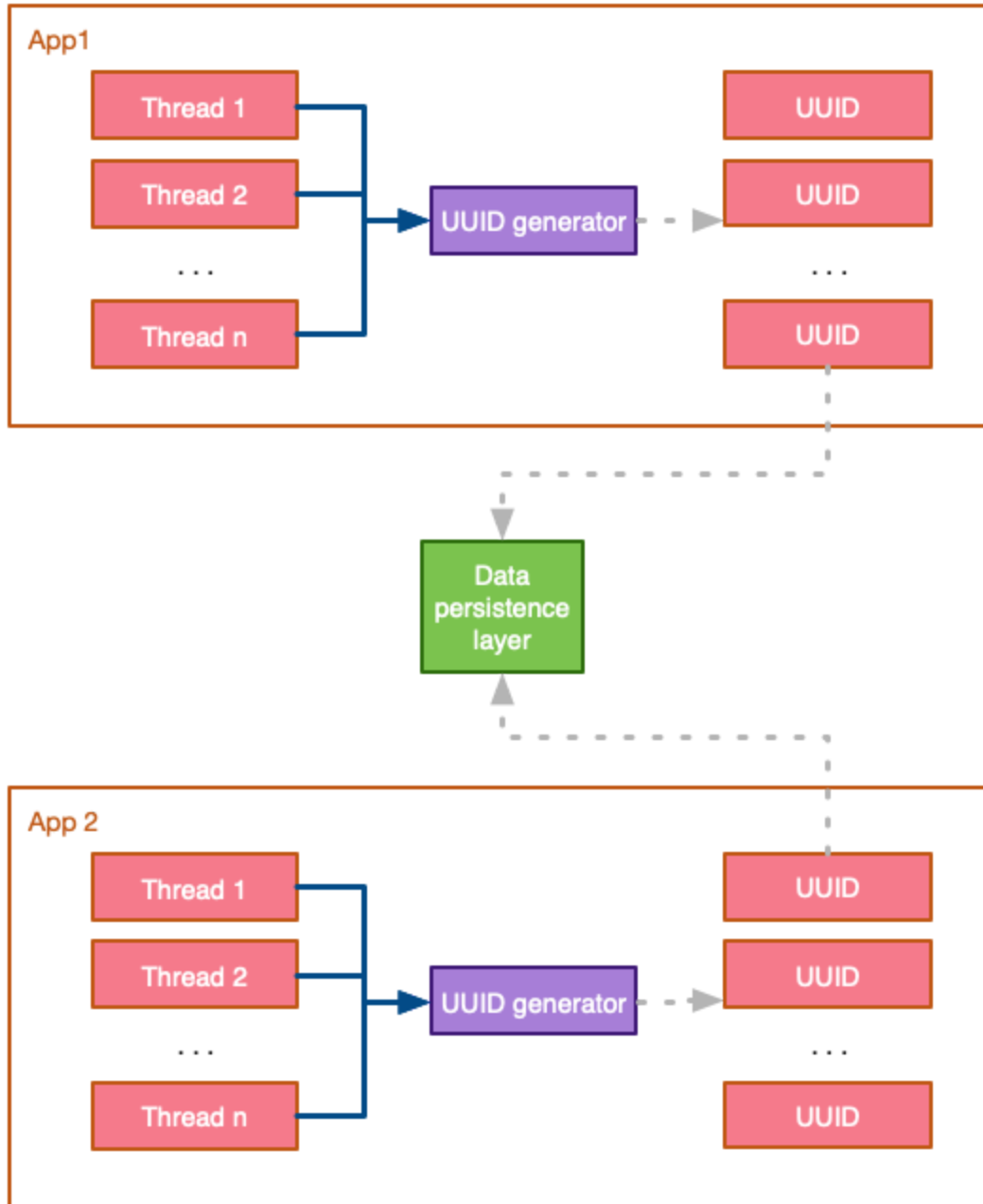
Dealing with 128-bit numbers is not the most developer-friendly way to depict information, so UUIDs are usually represented in a canonical text form where the 16 octets ( $16 * 8 \text{ bits} = 128 \text{ bits}$ ) are converted to 32 hexadecimal characters separated by hyphens, for a total of 36 characters:



**53d2d76e-5fa2-4738-ad35-16855d17bb33**

A sample of a v4 UUID (image by author)

As becomes apparent, the most interesting feature of UUIDs is that they can be generated in isolation and still guarantee uniqueness in a distributed environment. In addition, the underlying ID generation algorithm is not complicated nor does it require any synchronisation (at least, down to the 100-nanosecond level), so it can be executed in parallel:



Generating unique Ids in a distributed environment (image by author)

The intrinsic property of self-generation uniqueness make UUIDs one of the most frequently used ID-generation techniques in distributed environments. However, keep in mind that UUIDs require extra storage and may negatively affect your querying performance.

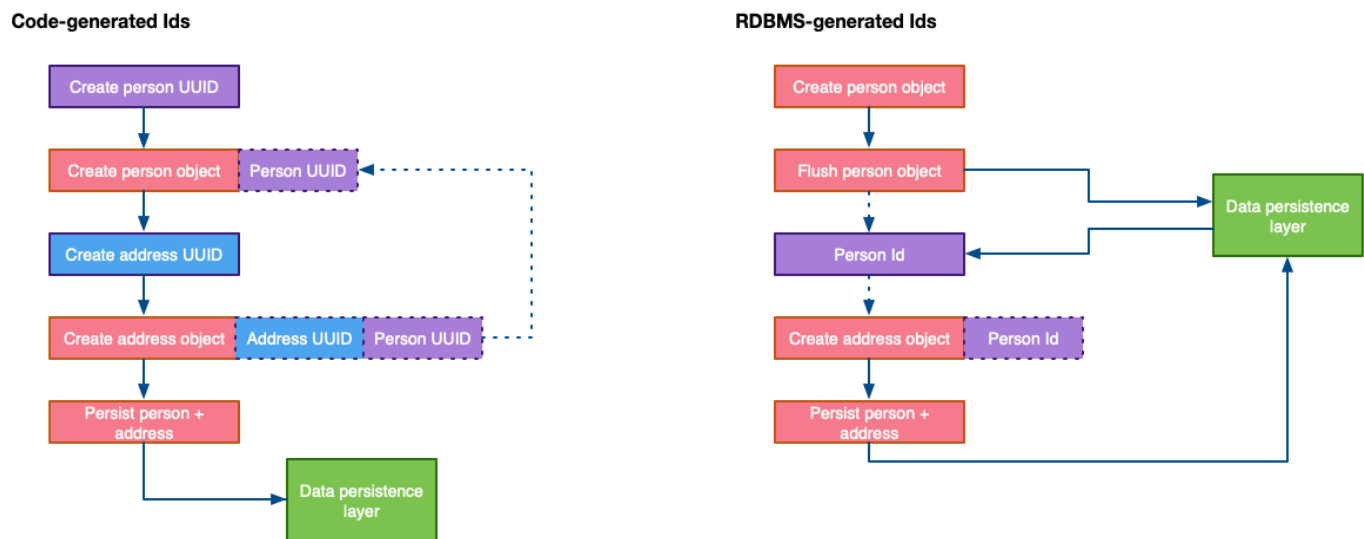
## Persistence Layer Generated IDs

A common approach when you don't want to generate unique IDs at the application level is to let your persistence storage take care of it. All recent RDBMS provide some sort of a column data type allowing you to delegate the generation of a unique identifier to them. MongoDB provides `ObjectID`, MySQL, and MariaDB provide `AUTO_INCREMENT`,

MS SQL Server provides `IDENTITY`, etc. The actual representation of the ID differs between different database implementation, however, the semantics about uniqueness remains the same.

Persistence layer generated IDs alleviate the problem of having to generate unique IDs in application code. However, if you operate a large database cluster with a very busy application in front, this approach may not be performant-enough for your needs.

Another practical issue when using persistence layer generated IDs is that the generated ID is not known to your code without a roundtrip to the database:



RDBMS vs code generate Id (image by author)

The above extra roundtrip to the RDBMS may slow down your application and can make your code look needlessly more complex, however, modern ORM frameworks can help you do this in a standardised way, irrespectively of the underlying RDBMS product you're using.

## ID servers, or Snowflake IDs

An ID server takes care of generating unique IDs for your distributed infrastructure. According to the implementation of your ID server, it can be a single server creating IDs or a cluster of servers creating high numbers of IDs per second.

Twitter needs no special introduction and with an average of 9000 tweets per second and peaks as high as 143199 tweets per second, they needed a solution that not only

scales across their vast infrastructure of servers but also generates IDs efficient for storage. This is how Twitter came up with the Snowflake project:

*Snowflake is a network service for generating unique ID numbers at high scale with some simple guarantees.*

Twitter was looking for a minimum of 10000 IDs per second per process with a response rate of <2ms. ID servers should require no network coordination at all between them and generated IDs should be, roughly, time-ordered. Finally, to keep storage to a minimum, IDs had to be compact.

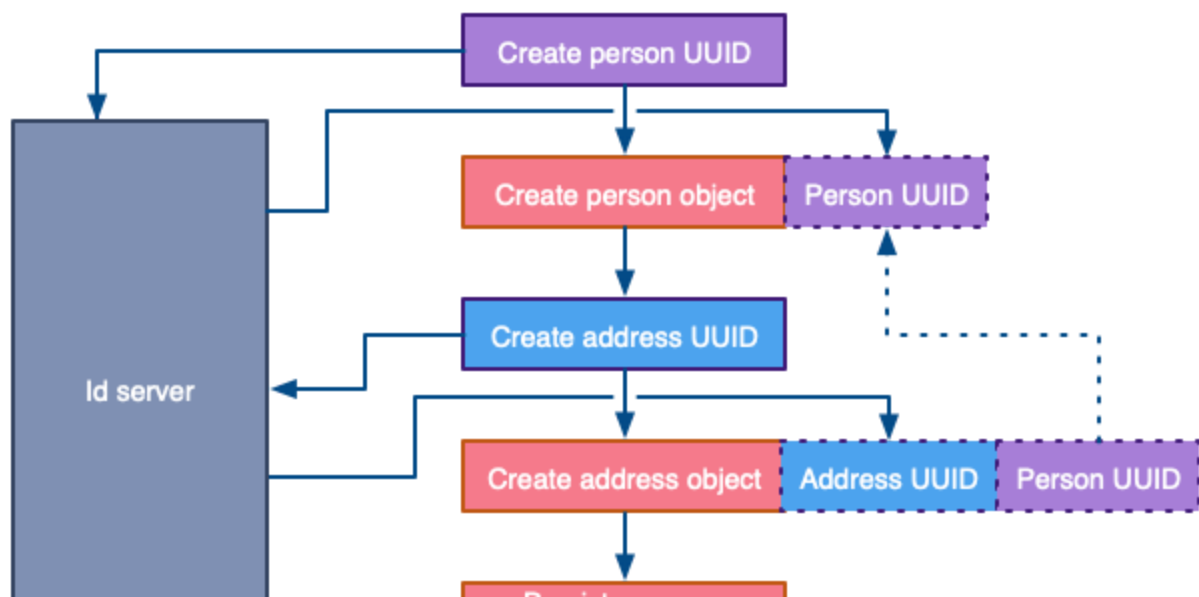
To solve the above project, Twitter developed the Snowflake project as a Thrift Server written in Scala. The generated IDs were composed of:

- Time — 41 bits (millisecond precision)
- Configured machine ID — 10 bits
- Sequence number — 12 bits — rolls over every 4096 per machine

Although Snowflake is now a retired Twitter project, replaced by a broader project, TwitterServer, the basic principles of how a distributed ID generator works still apply. Because of the independent nature of each generator, Twitter was able to scale its infrastructure as required, without introducing additional latency thanks to cluster synchronisation and coordination.

A solution with an ID server works in a similar way to the code-generated IDs:

### Id server-generated Ids





Id server-generated Ids (image by author)

As you may notice, performance is still degraded by the roundtrips to the ID server, however, this additional delay is considerably less than flushing an object to an RDBMS as it doesn't involve complex database operations.

An ID server provides a middle-ground solution, empowering you to control how and where your unique Ids are generated without introducing a complex, high latency-inducing infrastructure.

## Conclusion

Generating unique identifiers is a necessity for all applications that eventually need to persist data.

In this article three commonly used approaches were discussed: UUIDs — generating IDs locally, persistence layer driven IDs — centrally creating IDs, and Snowflake IDs — generating IDs as a network service.

Choosing a strategy for generating unique IDs in your application needs to look at your data as well as your persistence option and network infrastructure. As there is no one-size-fits-all solution, you should evaluate your options and choose the one aligned to your requirements and the scale you want to achieve.

Thank you for reading this article. I hope to see you in the next one.

Thanks to Zack Shapiro.

---

## Sign up for programming bytes

By Better Programming

A bi-weekly newsletter sent every Friday with the best articles we published that week. Code tutorials, advice, career opportunities, and more! [Take a look.](#)

Get this newsletter

[Programming](#)

[Software Engineering](#)

[Software Development](#)

[Software](#)

[Design Patterns](#)

[About](#) [Write](#) [Help](#) [Legal](#)

Get the Medium app

