Design Unique ID Generator

ASHISH PRATAP SINGH
APR 08, 2025 · PAID

Share

Any **distributed system** that operates at scale often relies on **unique ids.**

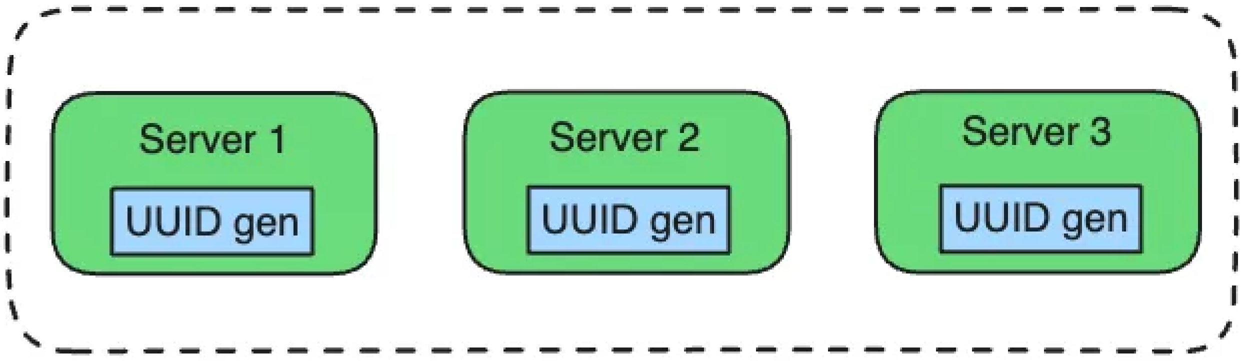
> For example, consider **order tracking in e-commerce**: each order placed by a customer is assigned a unique ID, allowing the system to track it through ever stage—order processing, payment, shipping, and delivery.

But how do we generate these IDs in a way that's fast, unique, reliable, and scala

In this article we'll dive into **7 popular approaches** to generate unique ids in distributed systems.

1. UUID (Universally Unique Identifier

UUIDs, also known as **GUIDs** (Globally Unique Identifiers) are **128-bit** numbers widely used for generating unique identifiers across distributed systems due to t simplicity and lack of dependency on a centralized source.



Visualized using Multiplayer

In this setup, each server can generate unique IDs independently.

UUIDs come in multiple versions:

1. **UUID v1** (**Time-Based**): Uses timestamp and machine-specific information the MAC address.
2. **UUID v3** (**Name-Based with MD5**): Generated by hashing a namespace and using MD5.
3. **UUID v4** (**Random**): Uses random values for most bits, providing a high deg uniqueness.
4. **UUID v5** (**Name-Based with SHA-1**): Similar to v3 but uses SHA-1 hashing stronger uniqueness.

The most commonly used version is **UUID v4.**

Format (UUID 4)

```
Example: 550e8400-e29b-41d4-a716-446655440000
```

- **Randomness** (**122 bits**): Most of the UUID is composed of random hexadeci digit `(0–9 or a–f)`.
- **Version** (**4 bits**): The third block's first character is always `4`, identifying it as version 4 UUID.
- **Variant** (**2-3 bits**): Located in the fourth block, it's either `8`, `9`, `a`, or `b`. It repre the variant and ensures that UUID follows the RFC 4122 standard.

Code Example (Python)

```
import uuid

# Generate a random UUID (version 4)
```

```
uuid_v4 = uuid.uuid4()
print(f"Generated UUID v4: {uuid_v4}")
```

Pros:

- **Decentralized**: UUIDs can be generated independently across servers.
- **Collision Resistance**: With 128 bits, UUID v4 has a collision probability so l practically negligible.

> To visualize: Even if every person on Earth generated 1 million UUIDs per sec it would take over 100 years to have a 50% chance of a single collision.

- **Ease of Implementation**: Most programming languages provide built-in libr for generating UUIDs.

Cons:

- **Large Size**: UUIDs consume 128 bits, which can be excessive for some stora sensitive systems.
- **Not Sequential**: UUIDs lack order, meaning they don't play well with indexi systems like B-Trees.

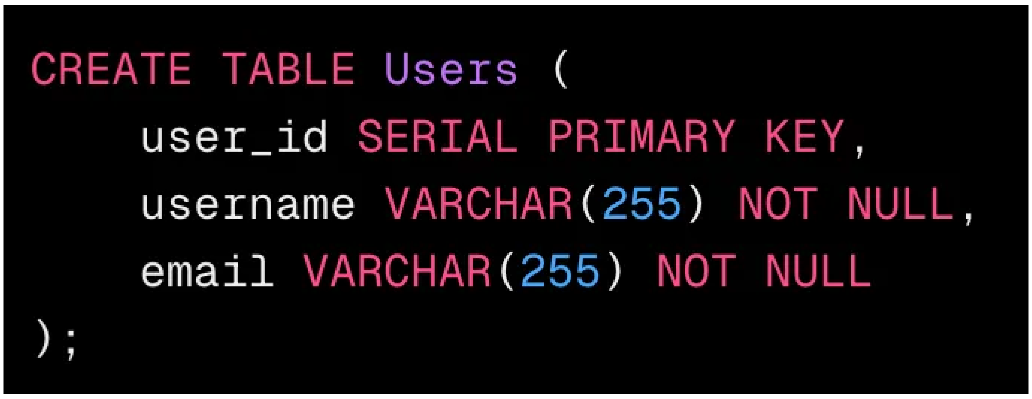
> UUIDs are ideal when you need globally unique IDs across distributed systen without central coordination and when order isn't important (e.g., Order IDs i commerce, Session IDs for User Authentication).

2. Database Auto-Increment

Database auto-increment is a feature in relational databases that automatically generates unique, sequential numeric IDs whenever a new record is inserted intc table.

Typically, the numbering starts from an initial value (often 1) and increments by amount (commonly 1) for each new row.

Example in SQL:



```sql
CREATE TABLE Users (
    user_id SERIAL PRIMARY KEY,
    username VARCHAR(255) NOT NULL,
    email VARCHAR(255) NOT NULL
);
```

Here, the `user_id` column will start from 1 and automatically increment for eacl row, generating unique values.

This approach works well for small applications with just one database node.

However, in distributed environments, depending on a single database node for I generation can quickly become a bottleneck and a single point of failure.

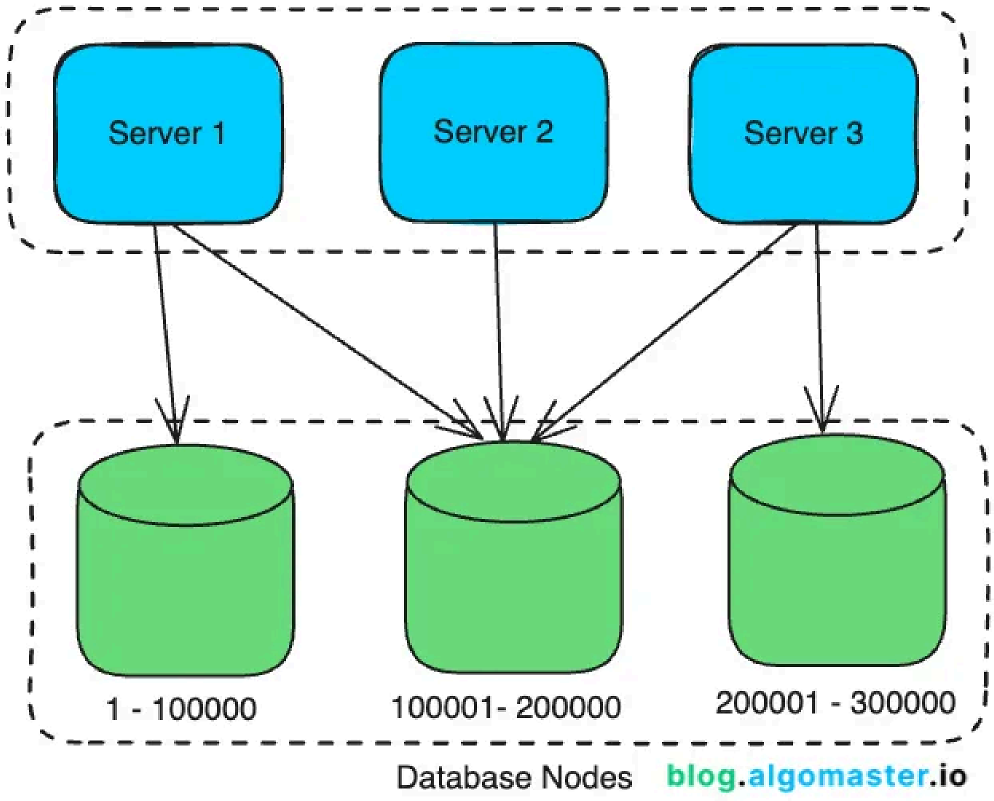
To make auto-increment work in distributed systems, here are two effective stra

2.1 Range-Based ID Allocation

In this approach, each **database node** is assigned a unique range of IDs, allowing to generate IDs independently and avoid conflicts or overlaps with other nodes.

For example, in a three-node setup:

- Node 1 can use IDs from 1 to 100000.
- Node 2 can use IDs from 100001 to 200000.
- Node 3 can use IDs from 200001 to 300000.



Limitations of Range-Based Allocation:

1. **Range Exhaustion**: High-traffic nodes may exhaust their assigned range qui requiring reallocation or range expansion.
2. **Complex Management**: As nodes are added or removed, reassigning and managing ranges can become complex.
3. **Waste of ID Space**: Uneven traffic across nodes may leave some ranges underutilized.

2.2 Step-Based Auto-Increment

In step-based auto-increment, each node generates IDs with a predefined step si

For example, if the step size is 3:

- Node 1 generates IDs as 1, 4, 7, 10, ….
- Node 2 generates IDs as 2, 5, 8, 11, ….
- Node 3 generates IDs as 3, 6, 9, 12, ….

This approach ensures each node generates unique IDs independently, but addir removing nodes may require reconfiguring the step size.

Pros of Database Auto-Increment

1. **Simplicity**: Straightforward to set up, as most relational databases support it natively.
2. **Sequential Order**: IDs are generated in a strictly increasing order, making it to sort records by insertion order.
3. **Low Storage Overhead**: IDs are typically small integers, making them efficie storage and indexing.

Cons of Database Auto-Increment:

1. **Coordination Overhead**: In a distributed setup, managing ranges or step increments requires careful setup and ongoing monitoring to avoid collision
2. **Predictable IDs**: Sequential IDs can be predictable, which may pose security in some applications (e.g., an attacker could guess the next ID).
3. **ID Exhaustion**: High insertion rates can exhaust the integer range, especiall smaller data types.

> Database Auto-Increment is useful when you need simple, sequential IDs (e.g relational database primary keys).

3. Snowflake ID (Twitter's Approach)

The **Snowflake ID** generation system, developed by Twitter, is a method for generating 64-bit IDs that are:

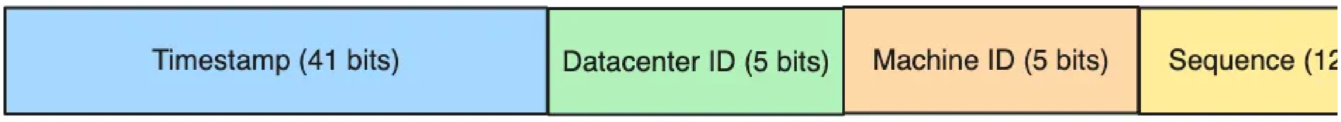
- Time-based
- Ordered

- Distributed-system friendly

It was created to handle the need for high-throughput, time-ordered IDs that ca horizontally across multiple data centers and machines.

These IDs are not only unique but also sequential within each machine, making highly efficient for indexing and ordering operations.

Format



Visualized using Multiplayer

```
Example Snowflake ID (binary):

000001101001100111001011001010100101010110001010111000000000000001
```

Breakdown:

1. Sign Bit (1 bit): Always set to `0` to ensure the ID is positive.

2. Timestamp (41 bits): The first 41 bits encode the timestamp in milliseconds si the Snowflake epoch (often set to November 4, 2010). This timestamp allows the be sorted chronologically.

3. Datacenter ID (5 bits): The next 5 bits represent the data center or region ID, allows for up to 32 (2^5) unique data centers.

4. Machine ID (5 bits): The following 5 bits represent the machine (or worker) IL within the data center, allowing for 32 machines per data center.

5. Sequence Number (12 bits): The last 12 bits are a sequence counter, which res every millisecond. This counter allows each machine to generate up to 4,096 (2^1 unique IDs per millisecond.

Pros

1. **Time-Ordered**: Snowflake IDs include a timestamp, making them naturally ordered by generation time. This is beneficial for indexing and time-series d
2. **Decentralized**: Each machine can generate unique IDs independently, withc requiring a central coordination server.
3. **High Throughput**: With 12 bits for the sequence, each machine can generat 4,096 unique IDs per millisecond, making Snowflake IDs suitable for high-tr environments.
4. **Compact and Efficient**: At 64 bits, Snowflake IDs are more storage-efficient UUIDs (128 bits).

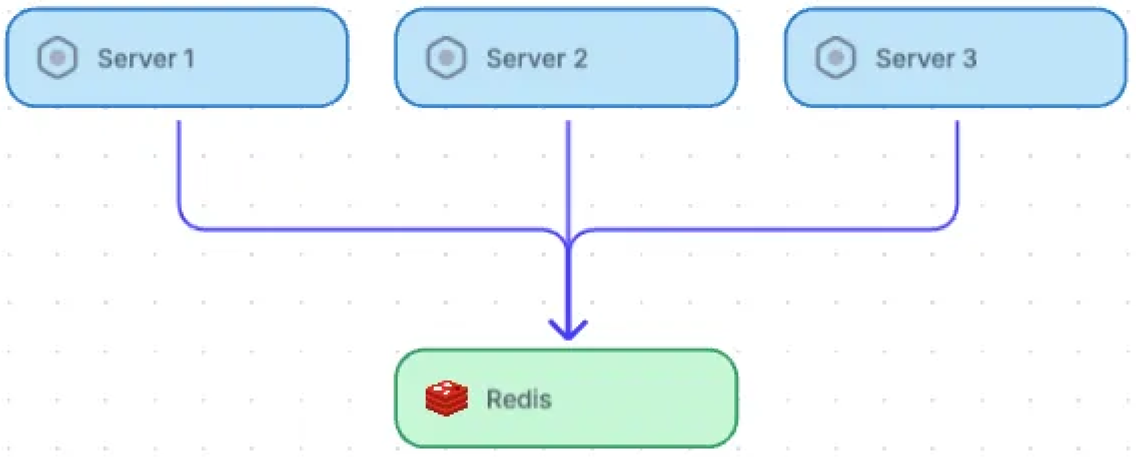
Cons

1. **Clock Synchronization**: Snowflake ID generation depends on synchronized clocks. If the system clock moves backward, it can lead to duplicate IDs or I generation errors.
2. **Limited Capacity**: Each machine can only generate up to 4,096 IDs per millisecond. If a higher rate is required, additional machines or other scalin solutions are needed.

> Snowflake IDs are ideal when you need unique, time-ordered IDs in distribut systems that require high throughput and scalability (e.g., social media posts, logs).

4. Redis-Based ID Generation

Redis, an in-memory key-value store, can also be used for ID generation due to i **atomic operations** and **low-latency** performance.



Visualized using Multiplayer

Here's how Redis-based ID generation works:

1. **Initialize a Key:** Setup a Redis key to store the current ID value.
2. **Increment on Demand**: Whenever a new ID is needed, an application node increments the counter using Redis's atomic `INCR` or `INCRBY` command.
3. **Return Unique ID**: The incremented counter value is guaranteed to be uniq it's returned to the application.

```
import redis

# Connect to Redis
client = redis.Redis(host='localhost', port=6379, db=0)

def generate_id():
    # Increment the key 'unique_id' and get the new value
    unique_id = client.incr('unique_id')
    return unique_id

# Generate a new ID
new_id = generate_id()
print(f"Generated ID: {new_id}")
```

Redis guarantees atomicity, so no two calls to `generate_id()` will ever receive t same ID, even if multiple nodes are concurrently accessing the Redis server.

Pros

1. **Atomicity**: Redis's `INCR` and `INCRBY` commands are atomic, ensuring each generated ID is unique and sequential without any risk of collision.
2. **High Throughput**: As an in-memory database, Redis provides very low laten making it ideal for high-speed ID generation.
3. **Simplicity**: Setting up Redis for ID generation is straightforward and requir minimal configuration.
4. **Sequential IDs**: Redis-generated IDs are sequential, making them suitable f ordered indexing in databases or applications where chronological order is important.

Cons

1. **Single Point of Failure**: Using a single Redis instance as the ID generator ca become a bottleneck and a potential single point of failure.
2. **Scalability Limitations**: While Redis can handle high throughput, using it a centralized ID generator limits horizontal scaling because every request dep on a single Redis instance.

> Redis-based ID generation is useful when you need high-speed, centralized II generation with sequential order, and the setup is primarily single-node.

5. Nano ID

NanoID is a small, URL-friendly, unique string ID generator designed for simpli flexibility, and performance in distributed systems.

Created as a modern alternative to UUID, Nano ID has gained popularity in fron applications and modern web development.

Unlike UUIDs, Nano ID doesn't follow a rigid structure, making it highly adapta
different applications.

Format

By default, NanoID generates a **21-character** ID using a **URL-safe** base64 alphab
Each character is randomly chosen from a 64-character set `(A-Z, a-z, 0-9, "`
`"_")`, creating a 128-bit identifier.

However, you can customize the length and character set to meet your applicatio
specific requirements.

```
Example: 7QLiKDgL-WG4E8z6xyVc0
```

Here's how to generate custom nano ids in Python:

```
import nanoid

# Generate default ID
id = nanoid.generate() # "V1StGXR8_Z5jdHi6B-myT"

# Custom length
custom_id = nanoid.generate(size=10) # "IRFa-VaY2b"

# Custom alphabet
custom_generator = nanoid.generate(
    alphabet='1234567890',
    size=6
)  # "123456"
```

Pros

1. **Compact and Readable**: With a default length of 21 characters, NanoID is sh
   and more readable than UUIDs, which are 36 characters.

2. **URL-Friendly**: NanoID uses a URL-safe character set by default, making it i for use in URLs without needing additional encoding.
3. **Decentralized**: Each node can generate unique IDs independently with mini risk of collision.
4. **Customizable**: You can adjust the length and character set to suit specific ne
5. **High Performance**: Nano ID's generation is fast, making it ideal for scenaric requiring rapid creation of many unique IDs.

Cons

1. **Non-Sequential**: NanoIDs are purely random and lack sequential ordering, v can lead to fragmentation in database indexes.
2. **Collision Probability**: Reducing the length of Nano ID increases the risk of collision, so longer IDs may be needed for critical applications.

> Nano ID is ideal for generating short, customizable, URL-friendly IDs in applications that don't require time-ordering, such as URLs, tokens, and front identifiers.

6. Hash-Based IDs

Hash-Based IDs are unique identifiers generated by applying **cryptographic has functions** to specific data inputs.

They're deterministic, meaning the same input always produces the same ID, ma them ideal for systems that need consistent identifiers, like **deduplication** and **caching**.

Format

The format of hash-based IDs depends on the hashing algorithm used, such as **M SHA-1**, or **SHA-256**.

These IDs are typically encoded as hexadecimal strings and can vary in length depending on the hash function:

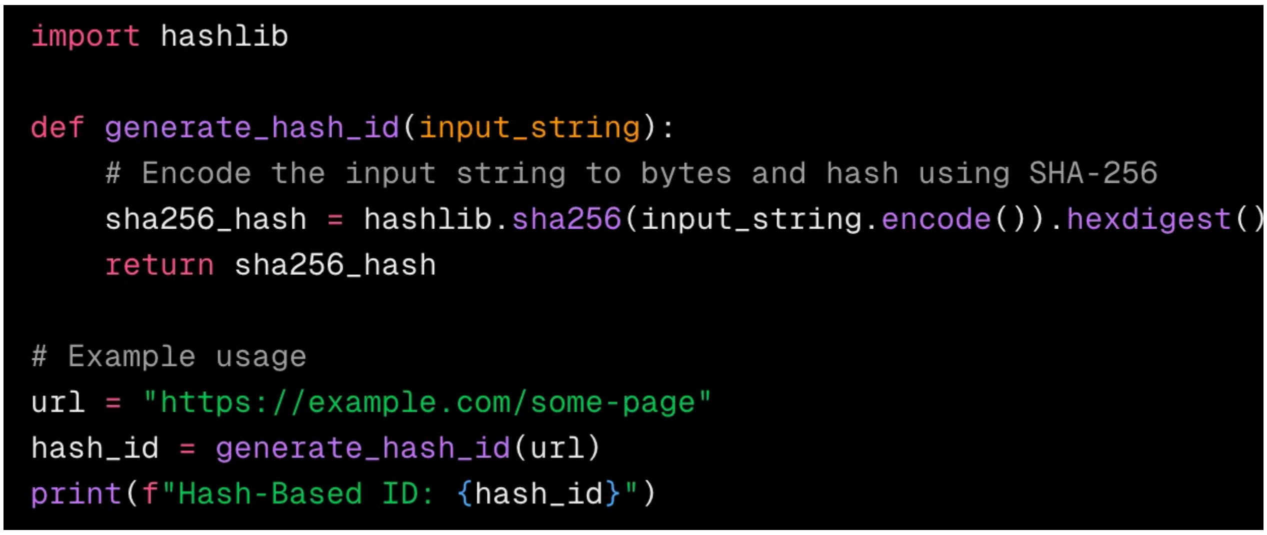
- **MD5**: 128 bits (32 hexadecimal characters)
- **SHA-1**: 160 bits (40 hexadecimal characters)
- **SHA-256**: 256 bits (64 hexadecimal characters)

The choice of hashing algorithm depends on the application's requirements for uniqueness, security, and collision resistance.

```
Example URL: https://example.com/some-page

SHA-256 hash output:
66e9c37ef3c04d3df238cd7d6b6b524f06c6e6dc9892e13c46f6d59f212dad0e
```

Code Example:



```python
import hashlib

def generate_hash_id(input_string):
    # Encode the input string to bytes and hash using SHA-256
    sha256_hash = hashlib.sha256(input_string.encode()).hexdigest()
    return sha256_hash

# Example usage
url = "https://example.com/some-page"
hash_id = generate_hash_id(url)
print(f"Hash-Based ID: {hash_id}")
```

Pros

1. **Deterministic**: The same input will always generate the same ID.
2. **Collision-Resistant**: Strong hash algorithms like SHA-256 provide high colli resistance, making it extremely unlikely for two different inputs to produce t same hash.

3. **Decentralized**: IDs can be generated independently across nodes in a distrib system without needing central coordination.

Cons

1. **Non-Sequential**: Hash-based IDs are non-sequential, which can lead to fragmentation in database indexes, slowing down query performance.
2. **Length**: Some hash functions, like SHA-256, produce long IDs (64 characters may be inefficient for certain applications.
3. **Collision Probability**: Using weaker hashes (e.g., MD5) increases the risk of collisions, which can cause issues in systems that require strict uniqueness.
4. **No Metadata**: Hash-based IDs are pure hashes and don't contain metadata information like timestamps or machine identifiers.

> Hash-Based IDs are useful when you need deterministic, unique IDs based on data, like content or URLs, rather than random values (e.g., deduplication, UR shorteners, caching systems).

7. ULID (Universally Unique Lexicographically Sortable Identifier)

A **ULID** is a 26-character, URL-safe string that combines:

- **Timestamp** (first 10 characters)
- **Randomness** (last 16 characters)

This format produces unique, readable and lexicographically sortable IDs.

Unlike UUIDs, which lack natural ordering, ULIDs embed a timestamp compon and use a compact, URL-friendly base32 encoding.

Format

```
01AN4Z07BY      79KA1307SR9X4MV3
|----------|    |----------------|
 Timestamp          Randomness
 10 chars           16 chars
 (48 bits)         (80 bits)
```

- **Timestamp (48 bits)**: The first 10 characters represent the timestamp in milliseconds since the Unix epoch (January 1, 1970). This allows ULIDs to be naturally sorted by creation time.
- **Randomness (80 bits)**: The remaining 16 characters are random, ensuring uniqueness even when multiple ULIDs are generated within the same millis

There are libraries available for generating ULIDs in many programming langua including JavaScript, Python, Java, and Go.

Pros

1. **Lexicographically Sortable**: ULIDs are time-ordered and naturally sortable, making them suitable for time-series data.
2. **Compact and URL-Friendly**: With 26 characters in base32 format, ULIDs ar shorter than UUIDs, making them suitable for embedding in URLs.
3. **Decentralized**: ULIDs can be generated independently on multiple nodes, as ID is based on a timestamp and a random component, reducing the need for centralized coordination.
4. **Readable and Error-Resistant**: The base32 encoding used in ULIDs is desig avoid confusing characters, such as "I" and "O".

Cons

1. **Limited Time Precision**: ULIDs use milliseconds for the timestamp, which not be precise enough for high-frequency systems that need IDs at microsec nanosecond levels.

2. **Limited Popularity**: Although gaining popularity, ULIDs are still less widel supported than UUIDs.
3. **No Embedded Metadata**: ULIDs encode only a timestamp; they do not inclu other metadata, such as machine ID or data center information, like Snowfla IDs.

> ULIDs are a great choice when you need unique, time-ordered, URL-friendly that can be generated independently without central coordination (e.g., time-s data, event logs).

Thank you for reading!

I hope you have a lovely day!

See you soon,
Ashish