Generate Unique IDs in Distributed Systems

1. UUID (Universally Unique Identifier)

- 128-bit numbers.
- 1. UUID v1 (Time-Based): uses timestamp and machine-specific info. like MAC address.
- 2. UUID v3 (Name-Based with MD5): generated by hashing a namespace and name using MD5.
- 3. UUID v4 (Random): Uses random values for most bits, providing a high degree of uniqueness.
- 4. UUID v5 (Name-Based with SHA-1): similar to v3 but uses SHA-1 hashing for stronger uniqueness.
- 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 hexadecimal digit (0-9 or a-f).
- Version (4 bits): The third block's first character is always 4, identifying it as a version 4 UUID.
- Variant (2-3 bits): Located in 4th block, it's either 8,9,a, or b. it represents the variant and ensure that UUID follows the RFC 4122 standard.
- **Pros**:
  - Decentralized: UUID can be generated independently across servers.
  - Collision Resistance: with 128 bits, UUID v4 has a collision probability so low it's practically negligible.
  - Ease of implementation
- **Cons**:
  - Not sequential: meaning they don't play well with indexing systems like B-Trees.

> UUIDs are Ideal when u need globally unique IDs across distributed systems without central coordination and when order isn't important .(session ID for user authentication)

2. Database Auto-Increment

- starts from 1, goes on increment by 1.

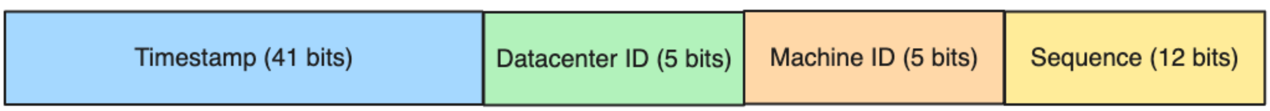
2.1 Range-Based ID allocation:

- each DB node is assigned a unique range of IDs, allowing them to generate IDs independently and avoid conflicts or overlaps with other nodes.
- Limitations:
  - if few nodes have higher traffic then their assigned range quickly gets exhausted.

2.2 Step-Based Auto-Increment:

- cons: predictable IDs

3. Snowflake ID (Twitter's Approach)

- 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 can scale horizontally across multiple data centers and machines.
- these IDs are not just unique but also sequential within each machine, making them highly efficient for indexing and ordering operations.
- 


```
Example Snowflake ID (binary):
0000011010011001110010110010101001010101100010101110000000000001
```

- the first 41 bits encode the timestamp in milliseconds since Snowflake epoch (often set to November 4, 2010). this timestamp allows the IDs to be sorted chronologically.
- the last 12 bits are a sequence counter, which resets every milliseconds, the counter allows each machine to generate up to (2^12) unique IDs per milliseconds.
- **Pros**:
  - Time-Ordered: Snowflake IDs include a timestamp, making them naturally ordered by generation time. beneficial for indexing and time-series data.
  - Decentralized
  - High Throughput: with 12 bits for sequence, suitable for high-traffic env.
  - Compact and more storage efficient (as 64 bits) than UUIDs(128 bits)
- **Cons**:

- Clock Synchronization:
    - Snowflake ID generation depends on synchronized clocks.
    - if System clock moves backward, it can lead to duplicate IDs or ID generation errors.

4. Redis-Based ID Generation:

- Redis, in-memory key-value store, can also be used for ID generation due to its atomic operations and low-latency performance.
- 1. Intialize 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 counter using Redis's atomic INCR or INCRBY command.
- 3. Return Unique ID: incremented counter value is guaranteed to be unique and it's returned to application.
- Redis guarantees atomicity, so no 2 calls to `generate_id()` will ever receive the same ID, even if multiple nodes are concurrently accessing Redis server.
- **Cons**:
- 1. Single Point of Failure: Using a Redis instance as ID generator can become a bottleneck and potential single point of failure.
- 2. Scalability Limitations: while Redis can handle high throughput, using it as a centralized ID generator limits horizontal scaling bcz every request depends on a single redis instance.

5. Nano ID

6. Hash-Based ID

- unique identifiers generated by applying cryptographic hash functions to specific data inputs.
- deterministic means the same input always produces the same ID, making them ideal for system that needs consistent identifiers, like deduplication and caching.
- Format:
    - hashing algo. used such as MD5, SHA-1 and SHA-256.
    - these IDs are typically encoded as hexademical strings and can vary in length depending on hash function:
        - `MD5`: 128 bits (32 hexadecimal chars)
        - `SHA-1`: 160 bits (40 hexadecimal chars)
        - `SHA-256`: 256 bits (64 hexadecimal chars)

- **Cons**:
  - No-metadata: hash-based IDs are pure hashes and don't contain metadata info. like timestamps and machine identifiers.

  > useful when u need deterministic, unique IDs based on input data, like content or URL, rather than random values (eg. deduplication, URL shorteners, caching systems).

7. ULID (Universally Unique Lexicographically Sortable Identifier)

- 26 chars, URL-safe string that combines:
- `Timestamp` (first 10 chars)
- `Randomness` (last 16 chars)
- This format produces unique, readable and lexicographically sortable IDs.
- ULIDs embed a timestamp component and use a compact, URL-friendly base32 encoding.

```
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 millisecond.
- **Pros**:
  - time-ordered and naturally sortable, making them suitable for time-series data.
  - with 26 chars in base32 format, ULIDs are shorter than UUIDs, making them suitable for embedding in URLs.

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