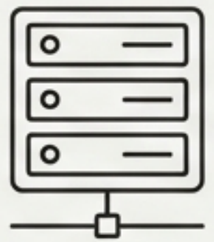
The Quest for the Perfect ID

A Guide to UUIDs vs. Snowflake IDs in Distributed Systems

It All Starts Simple: The Auto-Incrementing Primary Key

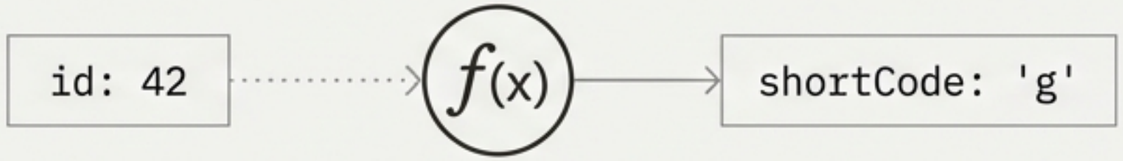
In a single database, life is easy. A `BIGINT` primary key with auto-increment gives us a unique, ordered identifier. We can even create a short code from it.

```
Primary key = auto-incrementing BIGINT
shortCode = Base62(id)
```



Monolithic Database

id (BIGINT)	url
1	example.com/a
2	example.com/b
3	example.com/c
4	...
5...	...

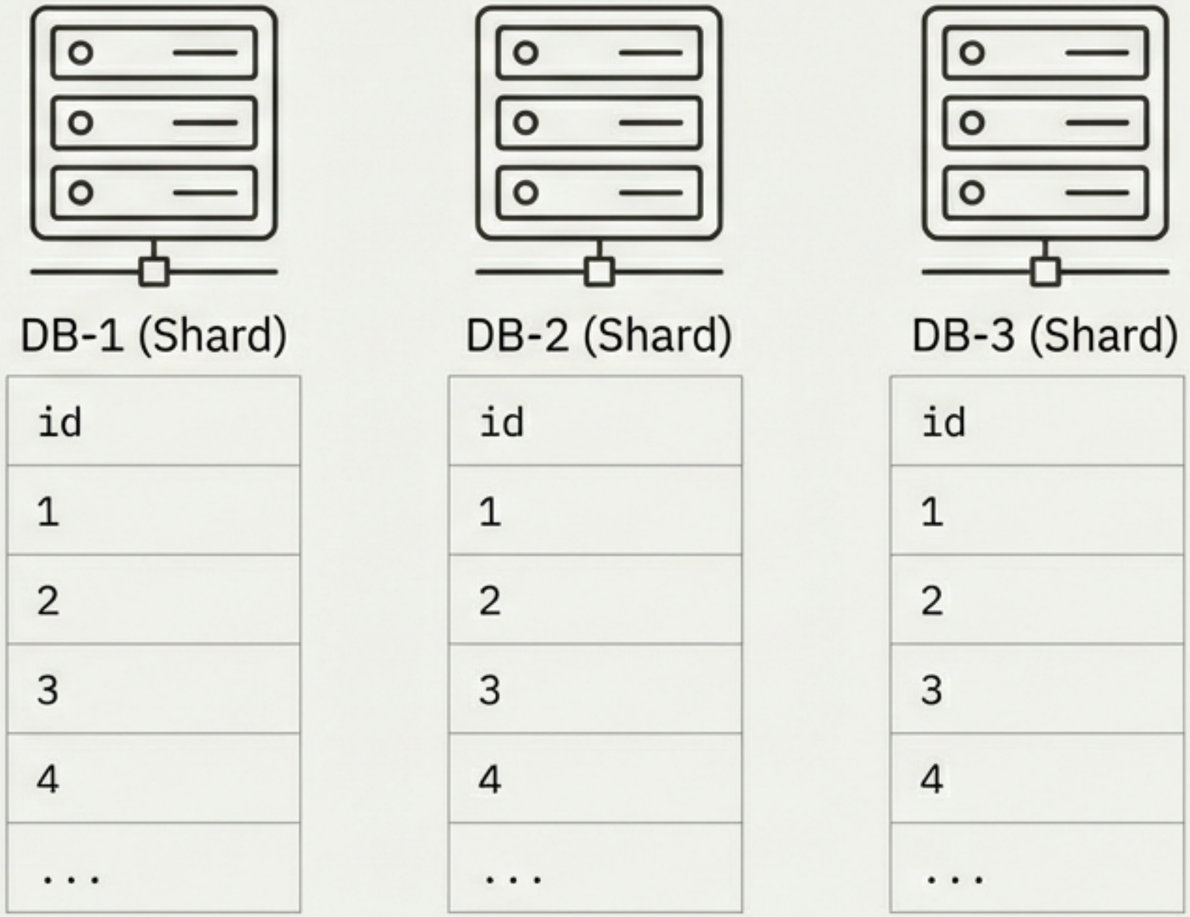


Base62

But Then, Scale Happens. And Our Simple World Breaks.

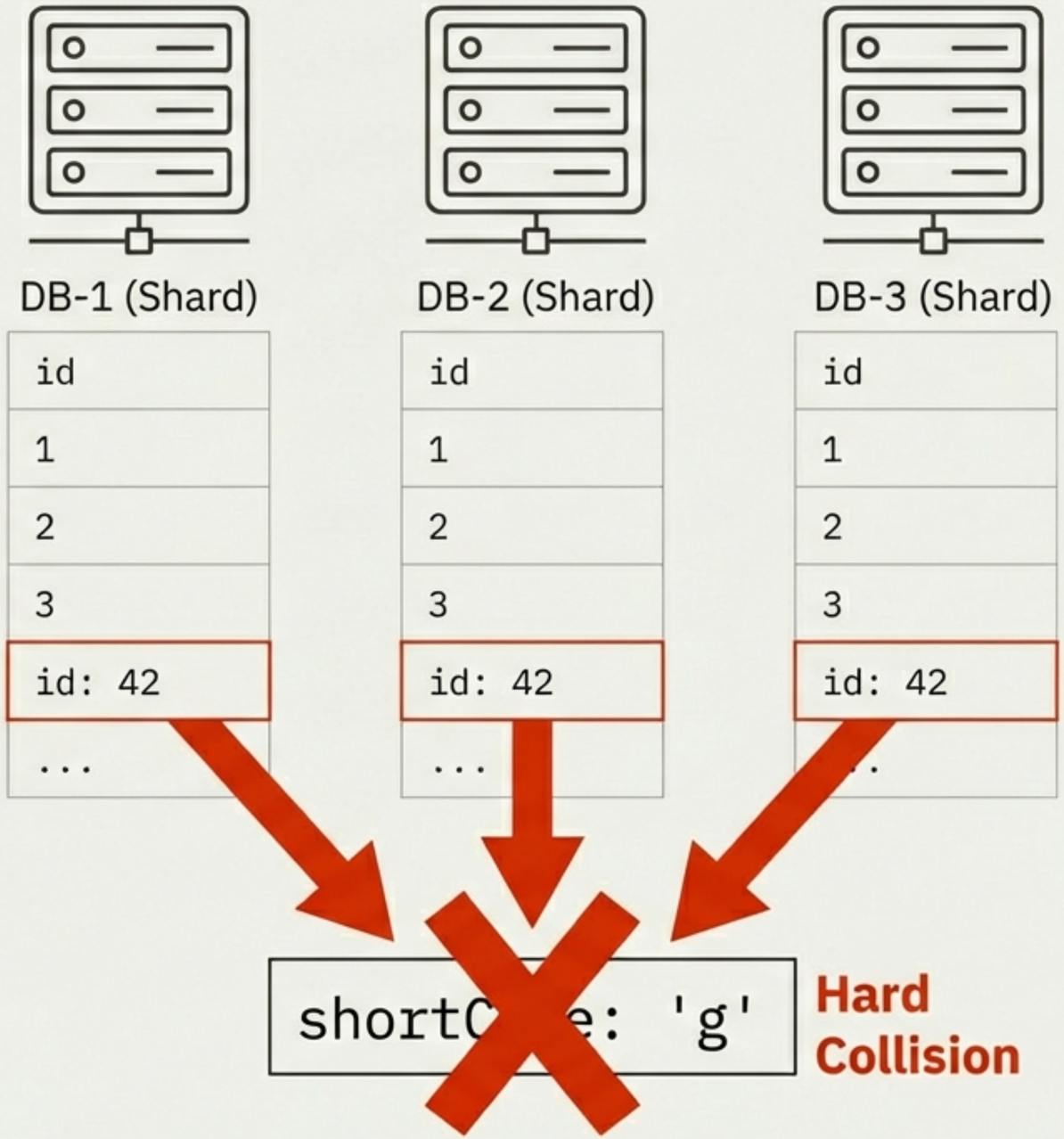
To handle more traffic, we shard our database. Now, each shard has its own independent auto-incrementing sequence.

This is a fundamental limitation of auto-increment IDs in distributed systems.



The Inevitable Collision

When different shards generate the same ID, they map to the same `shortCode`. The system becomes incorrect—linking a short URL to multiple, different original URLs.

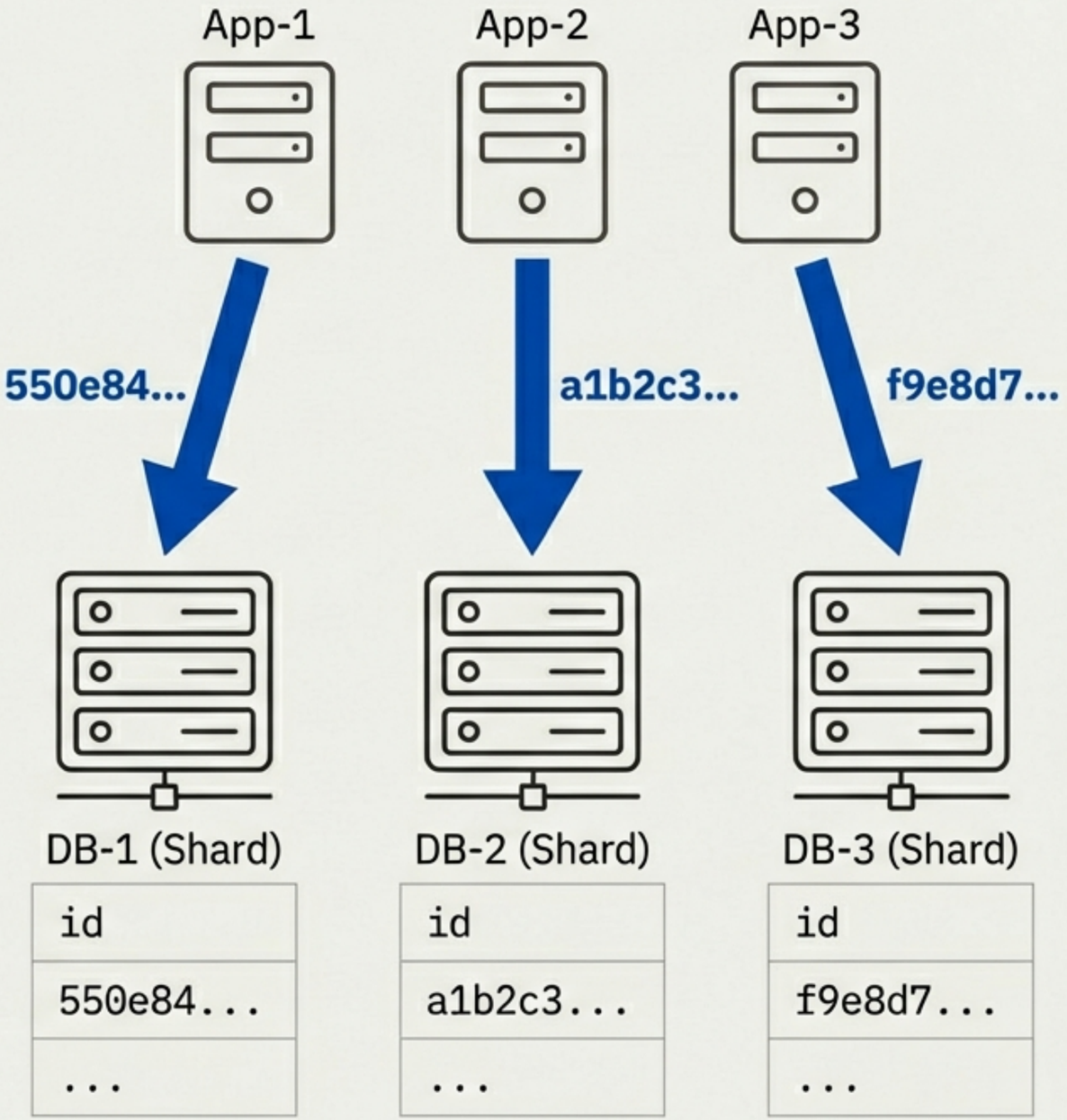


The First Solution: Guaranteeing Uniqueness with UUIDs

A UUID (Universally Unique Identifier) is a 128-bit number designed to be unique across machines, processes, and time without needing a central coordinator.

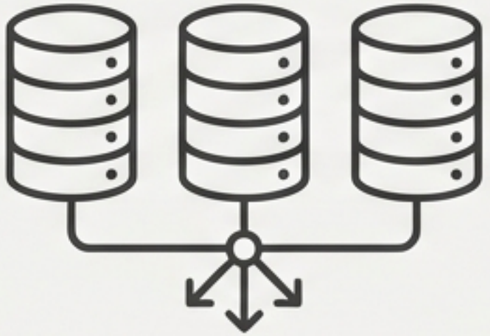
Example (UUID v4):
`550e8400-e29b-41d4-a716-446655440000`

Key Insight: UUIDs are generated at the **application layer**, not the database. There is no shared counter, and therefore, no collision risk.



Editorial Engineering

Why UUIDs Are a Powerful Choice

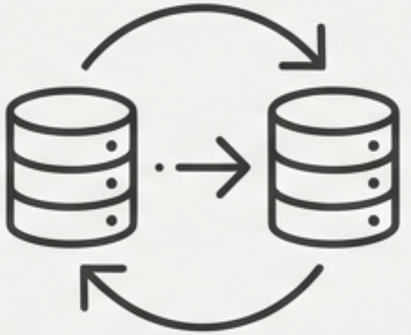


✅ No Collisions Across Shards

Safe for horizontal scaling and multi-region writes without a central ID generator.

✅ No Information Leakage

Random IDs don’t reveal creation order, total number of items, or business growth rate.



✅ Decouples Identity from Storage

You can reshard, migrate databases, or change storage engines without breaking existing IDs.

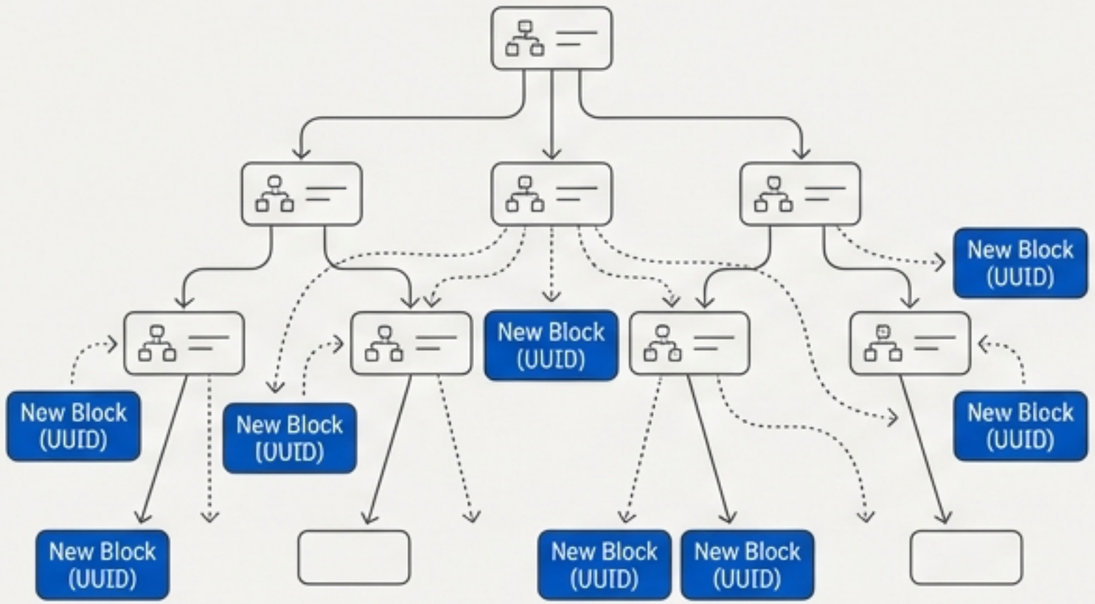
UUIDs are Correct, But They Are Not Ideal

UUIDs solve the critical correctness problem of **uniqueness in a distributed system. However**, they introduce significant operational and performance drawbacks, especially at high scale.

The Hidden Cost: Poor Database Index Locality

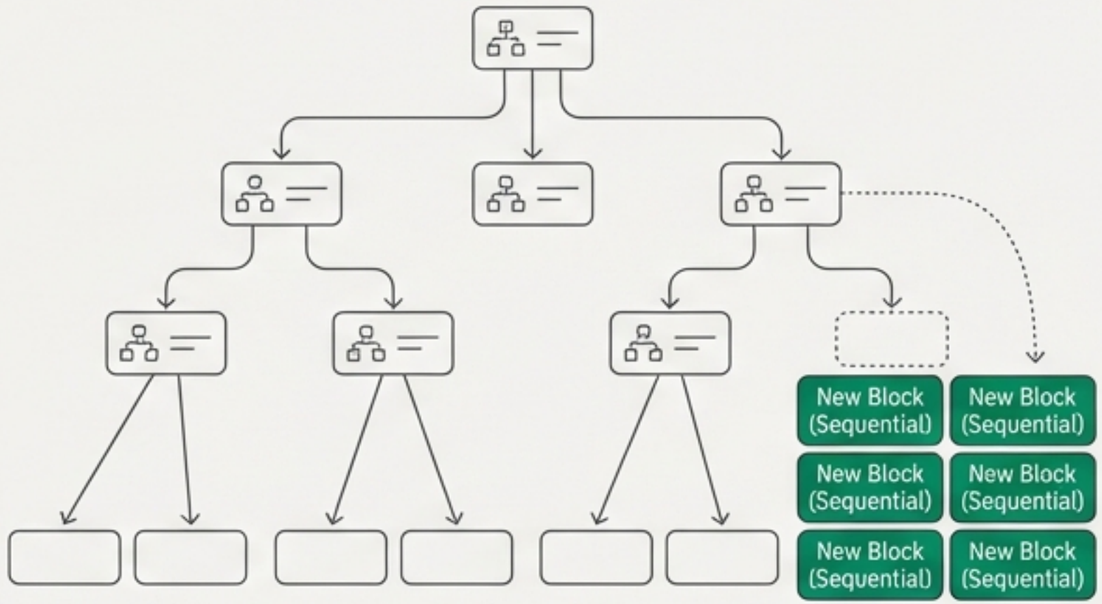
Random UUIDs (like v4) cause new records to be inserted at random locations within your database index. This leads to B-tree fragmentation and significantly slower writes at high throughput.

Random Inserts (UUIDs)



Metaphor: Throwing darts at a board.

Sequential Inserts (Ideal)



Metaphor: Stacking blocks.

More Practical Drawbacks of UUIDs

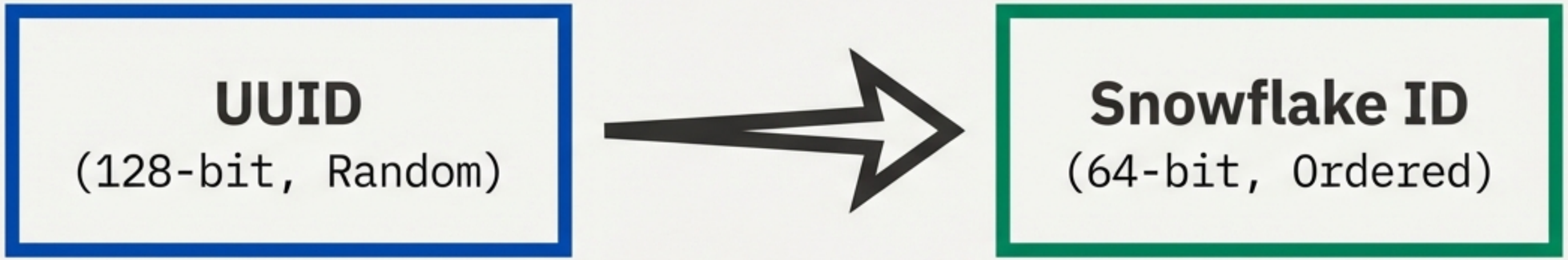
Beyond indexing, UUIDs introduce other challenges:

- ⚠ **Very Long Identifiers:** A 128–bit UUID Base62-encodes to ~22 characters. This is long for URLs and hurts usability.
- ⚠ **No Inherent Ordering:** You cannot sort by creation time using the ID alone. This requires an extra `` `created_at` `` column, which might not be on the primary index.
- ⚠ **Weak Observability:** A UUID tells you nothing about when or where it was created, making distributed debugging harder.

A Better Way: Snowflake IDs

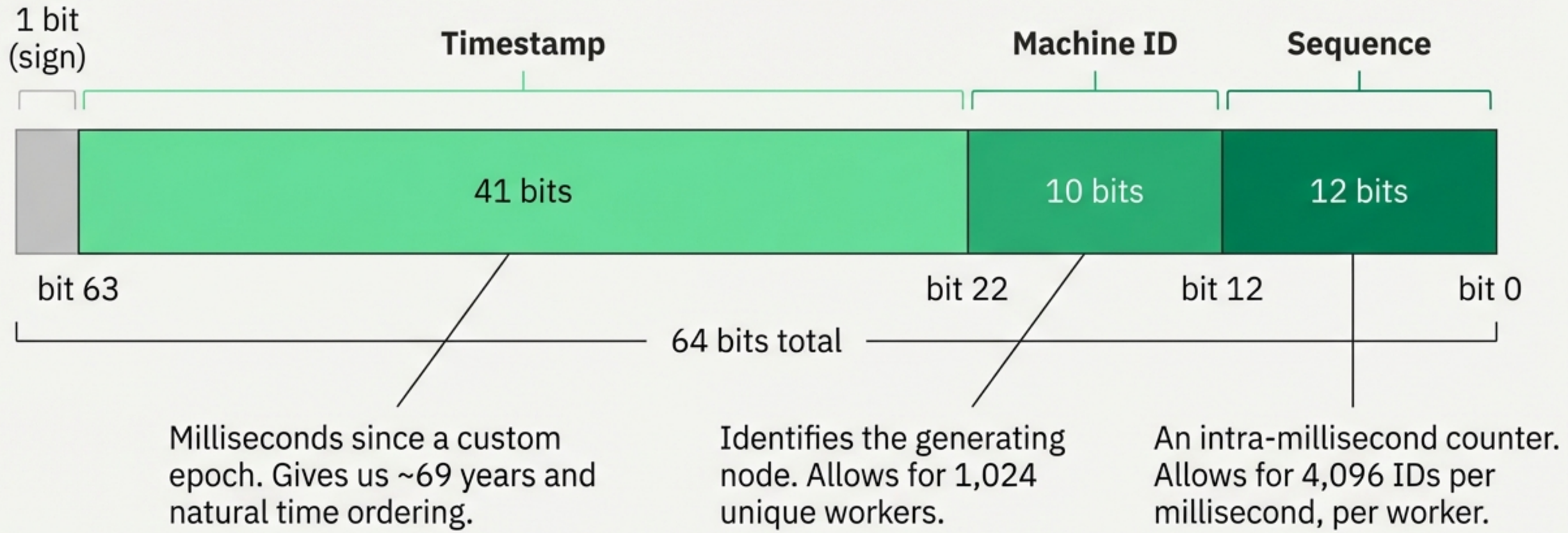
Snowflake IDs were created to fix the practical problems of UUIDs while preserving their global uniqueness. They are 64-bit integers optimized for:

- High write throughput
- Time ordering
- Shorter, URL-friendly identifiers



Dissecting a Snowflake ID

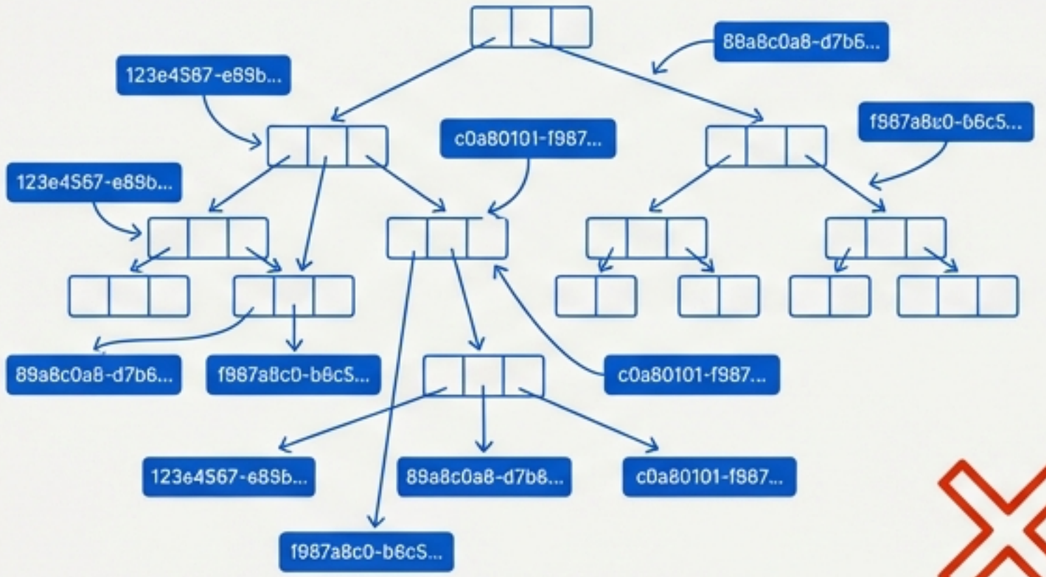
A Snowflake ID is a 64-bit integer composed of specific, meaningful parts.



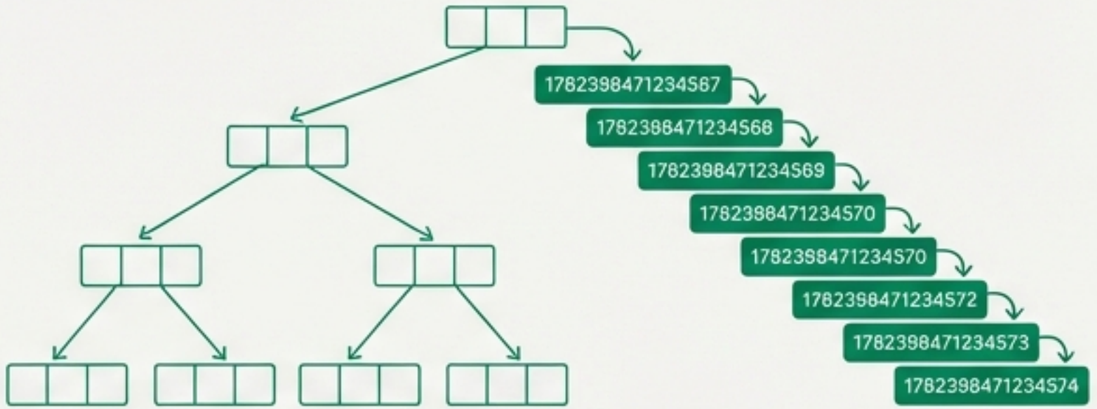
The Major Win: Time Ordering and Index Efficiency

Because Snowflake IDs are monotonically increasing, new records are almost always appended to the end of the index. This keeps the B-tree compact and makes writes dramatically faster at scale.

UUID Inserts



Snowflake Inserts



Solving the Other UUID Problems

Snowflake IDs are a comprehensive upgrade for high-throughput systems.

- **Shorter than UUIDs**: 64 bits vs. 128 bits. A Snowflake ID Base62-encodes to ~11 characters, half the length of a UUID.

  `s.io/4fT9XcQm2aJkP`

  From UUID (~22 chars)

  `s.io/LygHa16AHY`

  From Snowflake ID (~11 chars)

- **Built-in Traceability:** The ID itself contains the approximate creation time and the ID of the machine that generated it. This is invaluable for debugging distributed systems.

The Trade-Offs: Snowflake IDs Aren’t Free

While powerful, Snowflake IDs introduce operational complexity that UUIDs don’t have.

Operational Complexity:

You are now responsible for managing worker ID assignment and ensuring they are unique across your fleet.

Clock Skew Risk:

The system relies on synchronized clocks. If a clock moves backward, you risk generating duplicate IDs. Production systems must have safeguards for this.

Leaks Creation Time:

The timestamp is timestamp is part of the public ID. This is usually acceptable, but it is a form of information leakage.

UUID vs. Snowflake: The Final Showdown

Property	UUID v4	Snowflake
Global Uniqueness	✓	✓
Shard-Safe	✓	✓
Time Ordering	✗	✓
Index Friendly	✗	✓
URL Length	Long	Short
Ops Complexity	Low	Medium
Traceability	✗	✓

When to Use Snowflake IDs

You should choose a Snowflake-style ID when your system requires:

- * High scale and high write performance.
- * Sharded databases.
- * Sortable, time-ordered primary keys.
- * Compact, user-friendly identifiers (like short URLs).
- * Better observability and traceability from the ID itself.

This is why they are the choice for systems like Twitter, Discord, and Instagram.

A Simple Mental Model for Your Next System

UUID

→ Choose for **correctness first, simplicity first**. It's the default safe choice when performance is not the primary driver.

Snowflake

→ Choose for **scale, performance**, and **observability**. It is the professional tool for high-throughput distributed systems.

Snowflake IDs are what you use after you outgrow UUIDs.