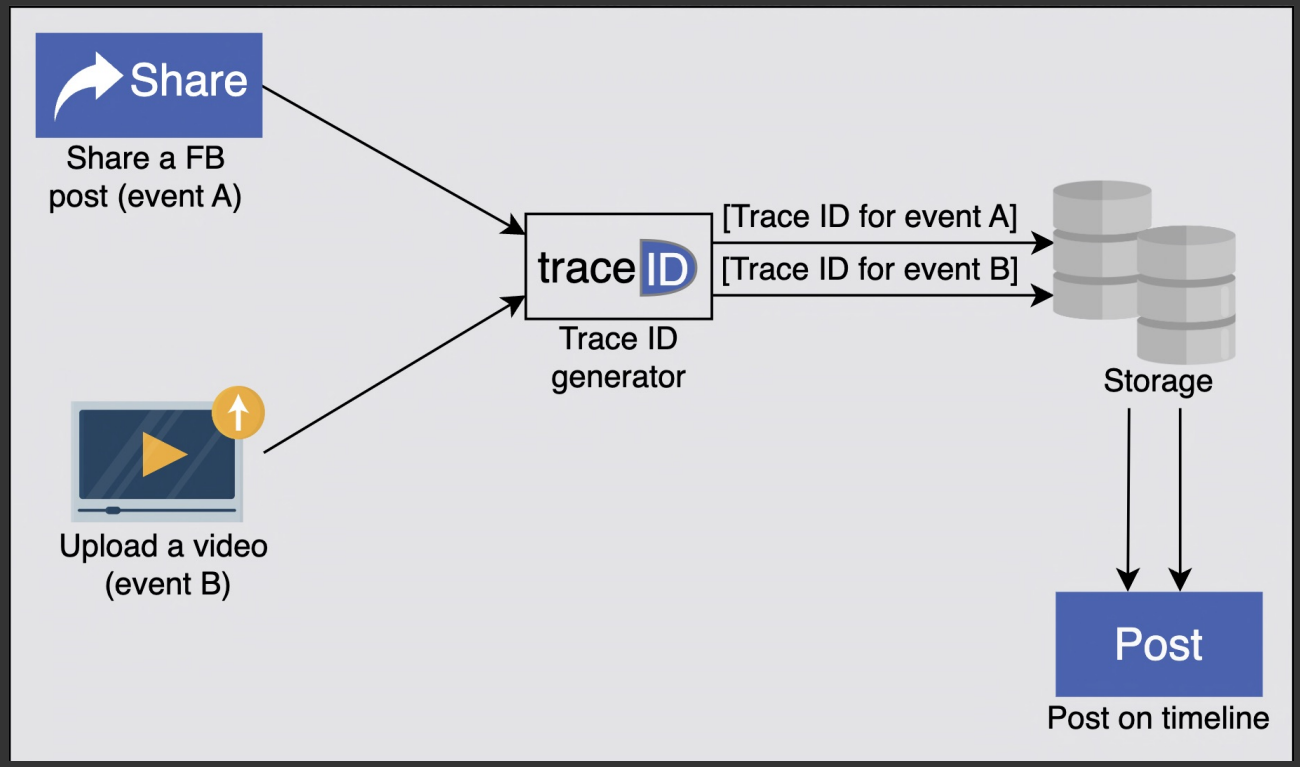
Sequencer

A unique ID helps us identify the flow of an event in the logs and is useful for debugging. A real-world example of unique ID usage is Facebook's end-to-end performance tracing and analysis system, Canopy. Canopy uses TraceID to identify an event uniquely across the execution path that may potentially perform hundreds of microservices to fulfill one user request.



Design a Unique ID Generator

Requirements for unique identifier

- **Uniqueness**: We need to assign unique identifiers to different events for identification purposes.
- **Scalability**: The ID generation system should generate at least a billion unique IDs per day.
- **Availability**: Since multiple events happen even at the level of nanoseconds, our system should generate IDs for all the events that occur.
- **64-bit numeric ID**: We restrict the length to 64 bits because this bit size is enough for many years in the future. Let's calculate the number of years after which our ID range will wrap around.

  Total numbers available = 2^{64} = 1.8446744 x 10^{19}

  Estimated number of events per day = 1 billion = 10^9

  Number of events per year = 365 billion = 365×10^9

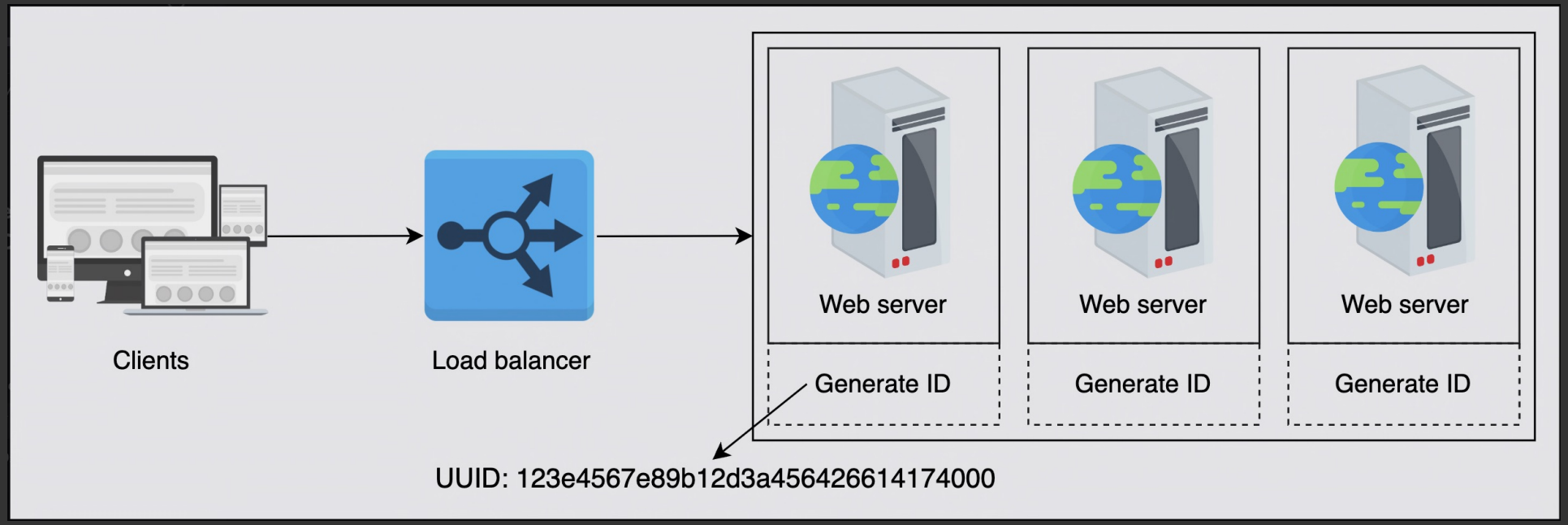
  Number of years to deplete identifier range = $\frac{2^{64}}{365 \times 10^9}$ = 50,539,024.8595 years

  64 bits should be enough for a unique ID length considering these calculations.

First Solution : UUID

A straw man solution for our design uses **UUIDs (universally unique IDs)**. This is a 128-bit number and it looks like $123e4567e89b12d3a456426614174000$ in hexadecimal. It gives us about 10^{38} numbers. UUIDs have different versions. We opt for version 4, which generates a pseudorandom number.

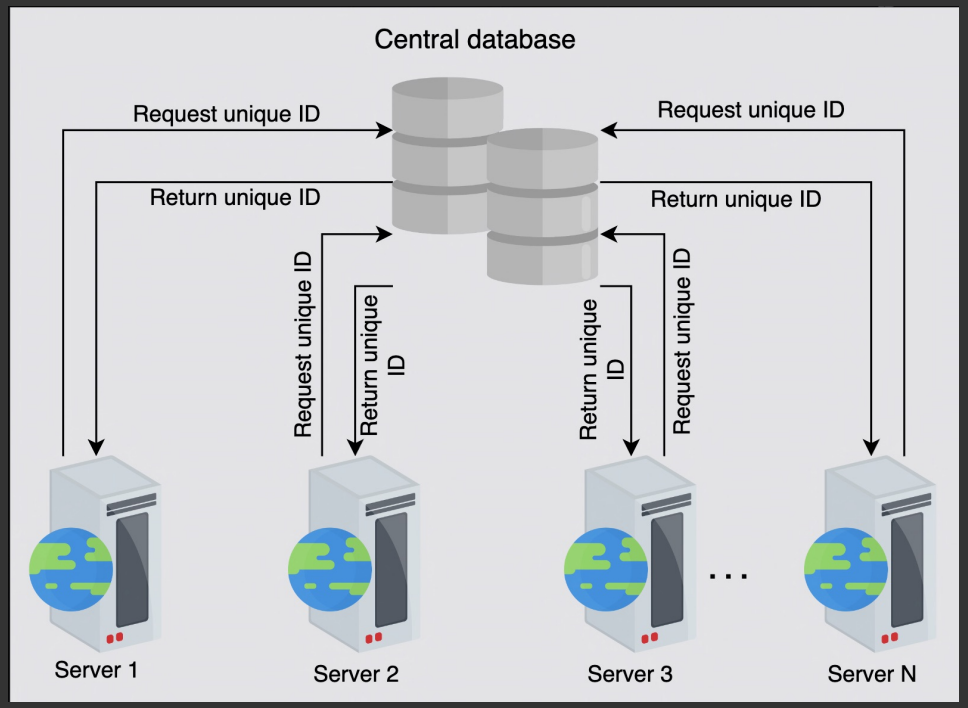
Each server can generate its own ID and assign the ID to its respective event. No coordination is needed for UUID since it's independent of the server. Scaling up and down is easy with UUID, and this system is also highly available. Furthermore, it has a low probability of collisions. The design for this approach is given below:



Requirements Filled with UUID

	Unique	Scalable	Available	64-bit numeric ID
Using UUID	✗	✓	✓	✗

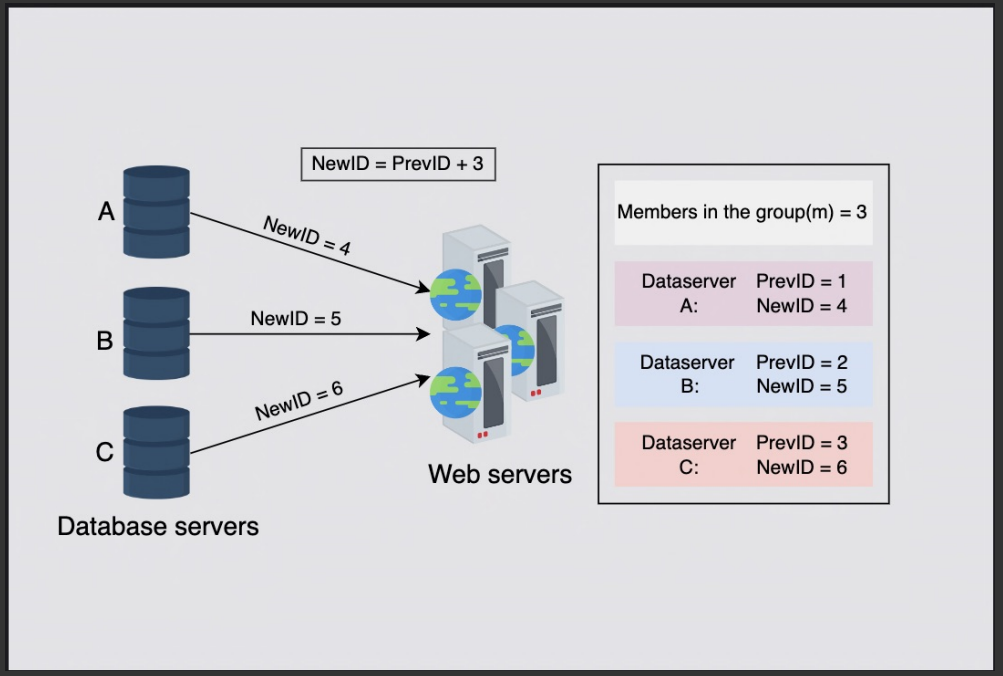
Second Solution : Using a Database



Let's try mimicking the auto-increment feature of a database. Consider a central database that provides a current ID and then increments the value by one. We can use the current ID as a unique identifier for our events.

What can be a potential problem of using a central database?

This design has a considerable problem: a single point of failure. Reliance on one database can severely affect a system. The entire system will stop working if the central database goes down.



To cater SPOF, we modify the conventional auto-increament feature increment by m.

Cons
Though this method is somewhat scalable, it's difficult to scale for multiple data centers. The task of adding and removing a server can result in duplicate IDs. For example, suppose m=3, and server A generates the unique IDs 1, 4, and 7. Server B generates the IDs 2, 5, and 8, while server C generates the IDs 3, 6, and 9. Server B faces downtime due to some failure. Now, the value m is updated to 2. Server A generates 9 as its following unique ID, but this ID has already been generated by server C. Therefore, the IDs aren't unique anymore.

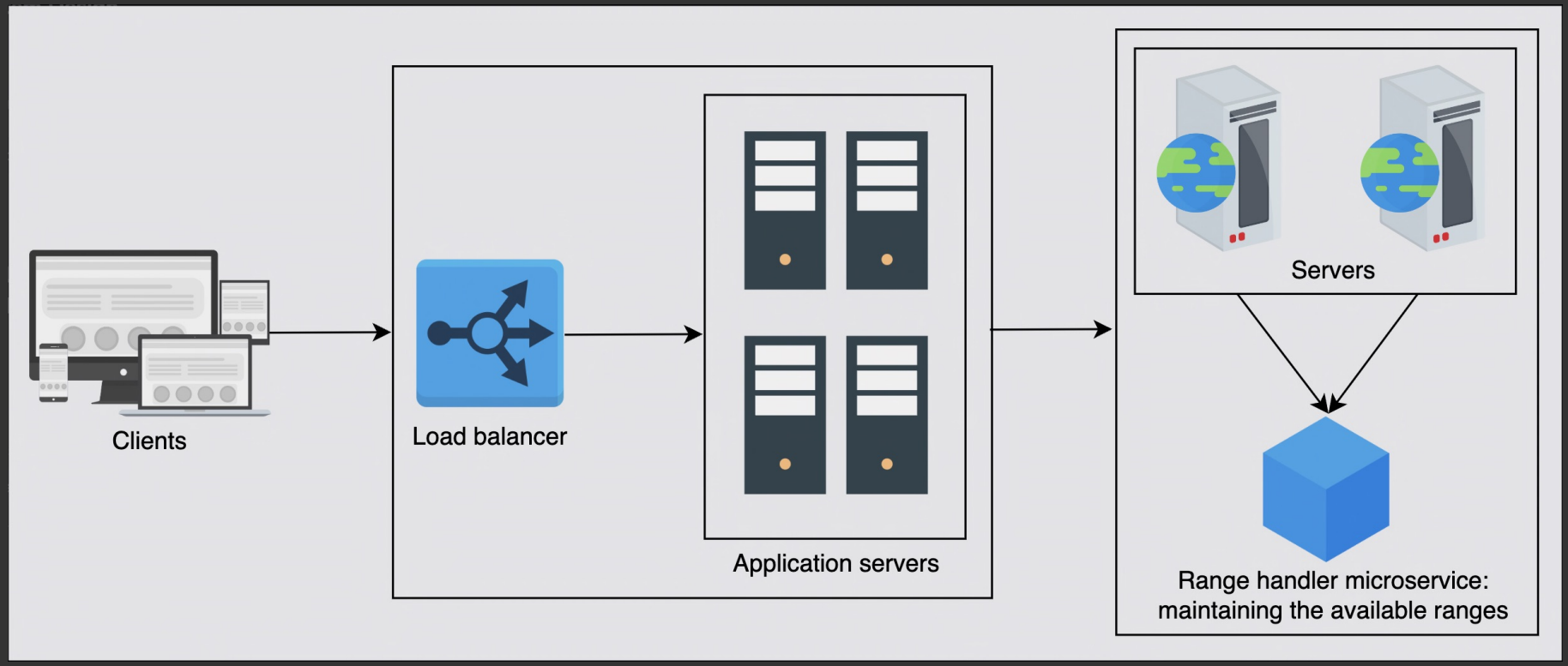
	Unique	Scalable	Available	64-bit numeric ID
Using UUID	✗	✓	✓	✗
Using a database	✗	✗	✓	✓

Third Solution: Using Range handler

We can use ranges in a central server. Suppose we have multiple ranges for one to two billion, such as 1 to 1,000,000; 1,000,001 to 2,000,000; and so on. In such a case, a central microservice can provide a range to a server upon request.

Let's say server 1 claims the number range 300,001 to 400,000. After this range claim, the user ID 300,001 is assigned to the first request. The server then returns 300,002 to the next user, incrementing its current position within the range. This continues until user ID 400,000 is released by the server. The application server then queries the central server for the next available range and repeats this process.

This resolves the problem of the duplication of user IDs. Each application server can respond to requests concurrently. We can add a load balancer over a set of servers to mitigate the load of requests.



Pros#
This system is scalable, available, and yields user IDs that have no duplicates. Moreover, we can maintain this range in 64 bits, which is numeric.

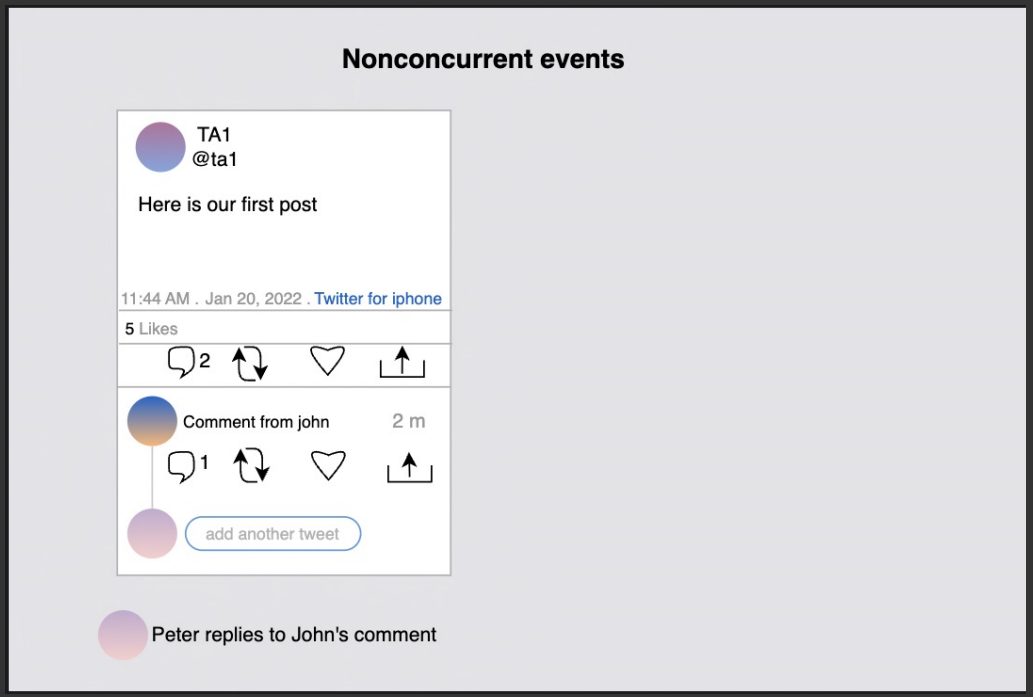
Cons
We lose a significant range when a server dies and can only provide a new range once it’s live again. We can overcome this shortcoming by allocating shorter ranges to the servers, although ranges should be large enough to serve identifiers for a while.

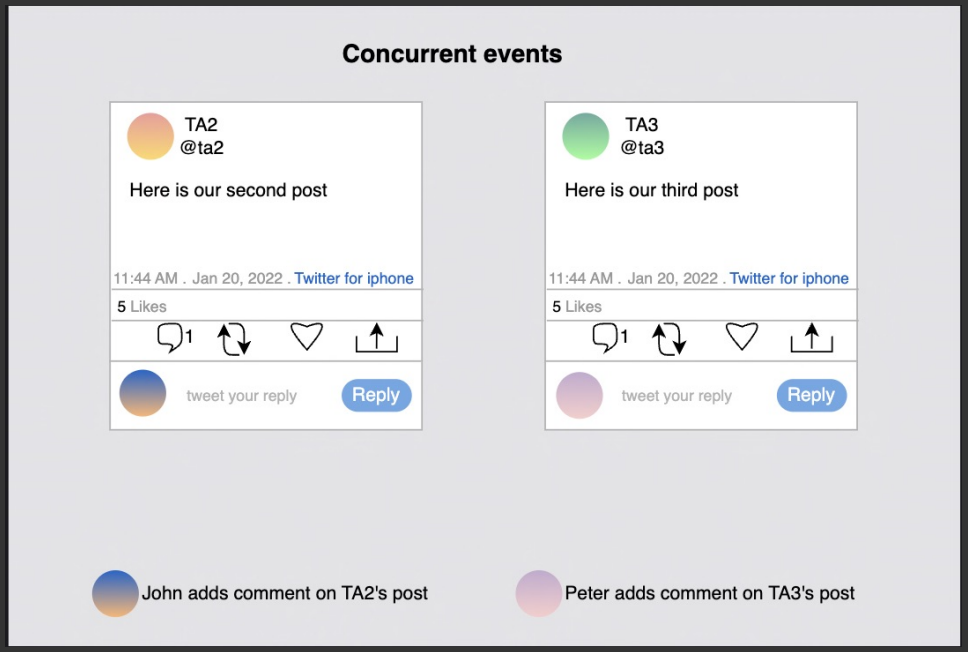
	Unique	Scalable	Available	64-bit numeric ID
Using UUID	✗	✓	✓	✗
Using a database	✗	✗	✓	✓
Using a range handler	✓	✓	✓	✓

Causality

we generated unique IDs to differentiate between various events. Apart from having unique identifiers for events, we're also interested in finding the sequence of these events.

We can also have concurrent events—that is, two events that occur independently of each other.





We can either use logical or physical clocks to infer causality. Some systems have additional requirements where we want event identifiers' causality to map wall-clock time. An example of this is a financial application that complies with the European MiFID regulations. MiFID requires clocks to be within 100 microseconds of UTC to detect anomalies during high-volume/high-speed market trades.

Physical clocks
There are often two types of physical clocks available in a computer: the time-of-day clock and monotonic counters.

The time-of-day clock

This usually has lower resolution in comparison to monotonic counters.
Network Time Protocol (NTP) can move the clock forward or backward, so it’s not always monotonic.
It may or may not incorporate leap seconds.

Monotonic counters

Monotonic counters usually have higher resolution than time-of-day clocks.
Monotonic counters should be used for the duration between two events rather than for the time.
These aren’t meaningful across different nodes. For instance, even on the same server with multiple processors, there can be a different counter per processor. The application needs to be careful when using counters from different processors.
The NTP might adjust it without violating monotonicity.
The NTP can only speed up or slow down the counter rate of change by up to 0.05%.

Reasons for clock drift

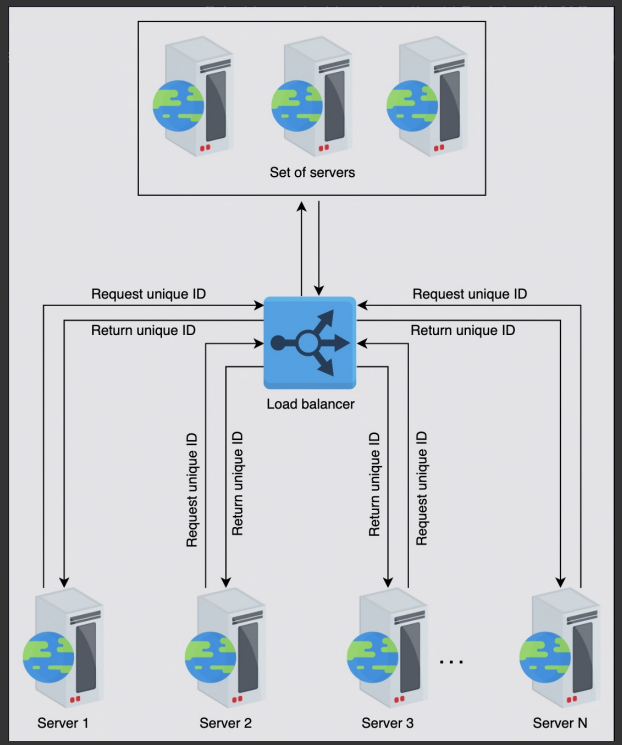
Physical clocks drift over time due to many reasons:

Temperature differences
The equipment’s age
Manufacturing defects
Virtualized clocks

Use UNIX time Stamps

UNIX time stamps are granular to the millisecond and can be used to distinguish different events. We have an **ID-generating server** that can generate one ID in a single millisecond. Any request to generate a unique ID is routed to that server, which returns a time stamp and then returns a unique ID. The ability to generate an ID in milliseconds allows us to generate a thousand identifiers per second. This means we can get $24(hour) * 60(min/hour) * 60(sec/min) * 1000(ID/sec) = 86400000 IDs$ in a day. That's less than a billion per day.

The ID-generating server is a single point of failure (SPOF), and we need to handle it. To cater to SPOF, we can add more servers. Each server generates a unique ID for every millisecond. To make the overall identifier unique across the system, we attach the server ID with the UNIX time stamp. Then, we add a load balancer to distribute the traffic more efficiently. The design of a unique ID generator using a UNIX time stamps is given below



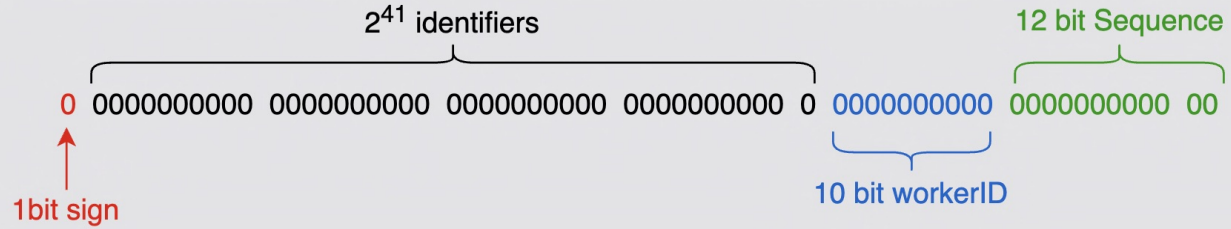
Pros
This approach is simple, scalable, and easy to implement. It also enables multiple servers to handle concurrent requests.

Cons
For two concurrent events, the same time stamp is returned and the same ID can be assigned to them. This way, the IDs are no longer unique.

	Unique	Scalable	Available	64-bit numeric ID	Causality maintained
Using UUID	✗	✓	✓	✗	✗
Using a database	✗	✗	✓	✓	✗
Using a range handler	✓	✓	✓	✓	✗
Using UNIX time stamps	✗	weak	✓	✓	weak

Twitter Snowflake

Let's try to use time efficiently. We can use some bits out of our targetted 64 bits for storing time and the remaining for other information. An overview of division is below:



Time to range depletion : $\frac{2^{41} \text{ identifiers}}{365 \text{ (days/year)} * 24 \text{ (hours/day)} * 60 \text{ (minutes/hour)} * 60 \text{ (seconds/minutes)} * 1000 \text{ (identifier/seconds)}}$ ~= 69 Years

Count of worker IDs: $2^{10} = 1024$

Count of sequence number combinations: $2^{12} = 4096$

Time to Range depletion ~ 69 years.

Pros
Twitter Snowflake uses the time stamp as the first component. Therefore, they're time sortable. The ID generator is highly available as well.

Cons
IDs generated in a dead period are a problem. The dead period is when no request for generating an ID is made to the server. These IDs will be wasted since they take up identifier space. The unique range possible will deplete earlier than expected and create gaps in our global set of user IDs.

Can you find another shortcoming in the design shown above?

Hide Answer
The physical clocks are unreliable. For such clocks, the error can be 17 seconds per day. If we measure time using these on a server, the time drifts away.

Considering a single server, we won't be affected by the drifting away of time since all transactions land on a single server. But in a distributed environment, the clocks won't remain synced.

Due to the unreliability of measuring accurate time, no matter how often we synchronize these clocks with each other or other clocks with accurate measurement methods, there will always be skew between the various clocks involved in a distributed system.

	Unique	Scalable	Available	64-bit numeric ID	Causality maintained
Using UUID	✗	✓	✓	✗	✗
Using a database	✗	✗	✓	✓	✗
Using a range handler	✓	✓	✓	✓	✗
Using UNIX time stamps	✗	weak	✓	✓	weak
Using Twitter Snowflake	✓	✓	✓	✓	weak

Using Logical clocks

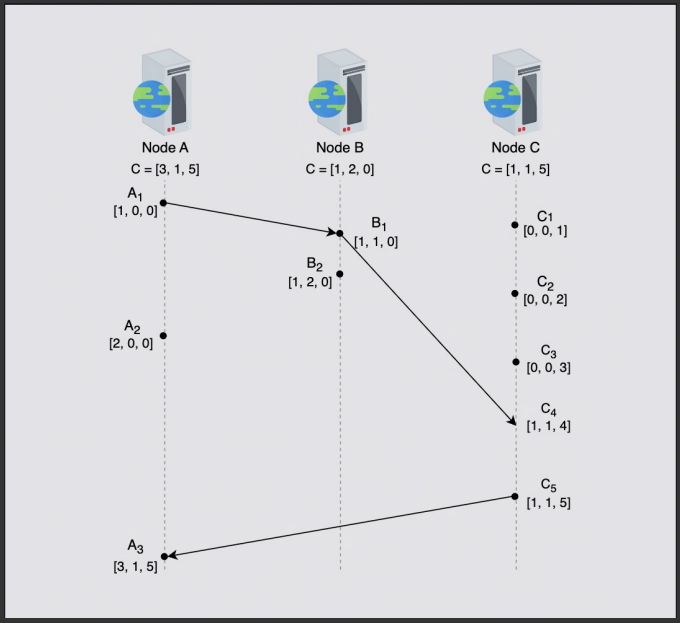
Lamport clock

In Lamport clocks, each node has its counter. All of the system's nodes are equipped with a numeric counter that begins at zero when first activated.

Lamport clocks provide a unique partial ordering of events using the happened-before relationship. We can also get a total ordering of events by tagging unique node/process identifiers, though such ordering isn't unique and will change with a different assignment of node identifiers.

Vector clocks

Vector clocks maintain causal history—that is, all information about the happened-before relationships of events. So, we must choose an efficient data structure to capture the causal history of each event.



	Unique	Scalable	Available	64-bit numeric ID	Causality maintained
Using UUID	✗	✓	✓	✗	✗
Using a database	✗	✗	✓	✓	✗
Using a range handler	✓	✓	✓	✓	✗
Using UNIX time stamps	✗	weak	✓	✓	weak
Using Twitter Snowflake	✓	✓	✓	✓	weak
Using vector clocks	✓	weak	✓	can exceed	✓

Would a global clock help solve our problem?

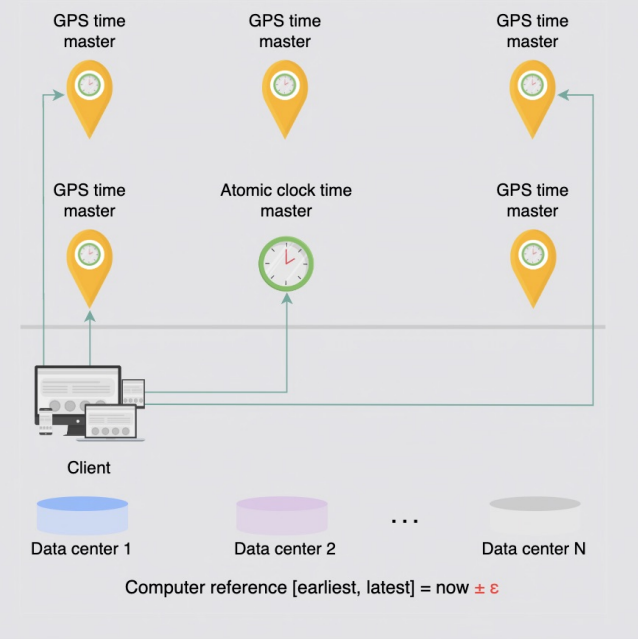
Since we don't have a global clock, even if each node can assign unique time stamps to the events happening, these time stamps will come from clocks running at different rates. This would make it harder to compare them, and they won't be unique.

However, if we have a global clock that gives us time upon request and is always accurate, then we can maintain the causality of events, as well as a unique ID. Such a clock would be significantly valuable, but time is tricky to handle in distributed systems.

True Time API

Google's TrueTime API in Spanner is an interesting option. Instead of a particular time stamp, it reports an interval of time. When asking for the current time, we get back two values: the earliest and latest ones. These are the earliest possible and latest possible time stamps.

Google deploys a GPS receiver or atomic clock in each data center, and clocks are synchronized within about 7 ms. This allows Spanner to keep the clock uncertainty to a minimum. The uncertainty of the interval is represented as epsilon.



The time reference will express a given interval as plus or minus epsilon

Pros

TrueTime satisfies all the requirements. We're able to generate a globally unique 64-bit identifier. The causality of events is maintained. The approach is scalable and highly available.

Cons

If two intervals overlap, then we're unsure in what order A and B occurred. It's possible that they're concurrent events, but a 100% guarantee can't be given. Additionally, Spanner is expensive because it ensures high database consistency. The dollar cost of a Spanner-like system is also high due to its elaborate infrastructure needs and monitoring.

	Unique	Scalable	Available	64-bit numeric ID	Causality maintained
Using UUID	✗	✓	✓	✗	✗
Using a database	✗	✗	✓	✓	✗
Using a range handler	✓	✓	✓	✓	✗
Using UNIX time stamps	✗	weak	✓	✓	weak
Using Twitter Snowflake	✓	✓	✓	✓	weak
Using vector clocks	✓	weak	✓	can exceed	✓
Using TrueTime	✓	✓	✓	✓	✓