Problem Statement

Design a system where messages consumed by your system should preserve the order of the messages coming in such that older messages do not overwrite newer messages.

Assumptions

- I am assuming that we are concerned about the order of messages, as seen **at arrival time** into our distributed service. We are not concerned with any re-ordering that happens in the WAN due to varying queue sizes in any in-path devices. Furthermore, we are also not worrying about any re-ordering that may happen between the load-balancer and the receiving service (although we don't really expect any measurable delay/re-ordering to happen at that level).

- We are using the system timestamp as the authoritative time source to generate the event Timestamp. We need to acknowledge that this is not a perfect system, and even using NTP there will be an inevitable desynchronization between separate instances of the service. However, this will likely be inconsequential as it will be of the order of ~ms in most cases.
  This is not a problem we can solve easily, and more importantly - we shouldn't have to. If we're worried about re-ordering due to a 50us timestamp difference, we should really be worrying about all the re-ordering and delays that happens in the network as packets potentially take different routes to reach our load-balancers. What happens if the client issues an update to the same event (id) 1ns after the original event?

- As I am not aware of the entire problem domain, we need to assume that partial updates are to be allowed. I will assume that any field outside of ID and Timestamp is allowed to be NULL. This does not change any proposed solution in a meaningful way, if we allow NULLs to be present on INSERT and modify our UPDATE queries to check if each field is NOT NULL before executing that statement. In other words, it simply makes the existing queries more verbose.

- I have chosen the rdbms to be MySQL, however that's an arbitrary choice that only affects non-standard SQL syntax in the case of conditional UPDATEs. This can also easily be done with PostGres (and any other major SQL database).

- I make **NO ASSUMPTIONS** as to the **CONSISTENCY** requirements of this problem. As such, I am proposing several solutions that will use both a **SYNCHRONOUS (Strong Consistency)** and **ASYNCHRONOUS(eventual consistency)** approach.

Motivation

Preserve the order of events arriving into our system to satisfy a requirement of our API. Clients of the API can assume that as long as there's a non-trivial difference in time between each update, our API will reflect the latest state.

Alternatives

The solutions are broken down into 2 categories. SYNC solutions assume a strong consistency requirement, and ASYNC solutions allow for eventual consistency in our state.

- SYNC1 - conditional updates in our UPDATE clause
- SYNC2 - SELECT for UPDATE transaction that allows for pre-emptive row locking (db-agnostic SQL)

- ASYNC1 - event sourcing system with incoming "raw" events stored in a separate table
- ASYNC2 - event sourcing system with events stored in Kafka

Limitations and Trade-offs

These will be discussed on a per-solution basis. Please see the individual design details documents for each proposed solution.

Decision

As the simplest and least time-consuming approach to immediately addressing this problem, we can start by implementing solution **SYNC1**. This is the least performant/scalable solution, however at lower traffic loads it should be perfectly adequate to satisfy the requirements of this design.

If we find that performance is **NOT SATISFACTORY,** we can move to one of the **ASYNC** solutions. This will allow us to massively limit the amount of writing (and row locking) to our primary events table. The **ASYNC2** solution allows us to have an almost arbitrary horizontal scaling capability - as we can add an arbitrary number of partitions (each with their own consumer/reader).