Strong Consistency Solutions

Solution SYNC1 (MySQL specific)

This solution retains the original architecture shown in the problem statement, with a load-balancer fronting a distributed service that processes incoming “event” records.

The only difference is that instead of allowing the database to be timestamping these events (likely by means of an INSERT/UPDATE trigger - since we do not see this field in the provided SQL), we are timestamping the events at arrival time (at each service instance) and then using that timestamp in the query.

In addition, we’re adding a condition in the UPDATE part of the SQL query that runs for each incoming event record.

Below is the modified SQL, it is in MySQL-specific syntax:

```
INSERT INTO table(id,data,enable,timestamp) VALUES (@id, @data, @enable, @timestamp)
ON DUPLICATE KEY UPDATE
timestamp= IF(timestamp < @timestamp,  @timestamp, timestamp),
data= IF(timestamp < @timestamp,  @data, data),
enable= IF(timestamp < @timestamp,  @enable, enable);
```

This will either insert a new record for a new event ID, or update the record if the existing timestamp is older than the incoming one.
This does not require an additional index in the table, as we are only searching by primary key (ID).

Trade-offs & Limitations:

- This approach can cause queries to block often if certain message IDs are often being updated, potentially limiting the performance of the overall system.

- In a pathological scenario, where a client keeps consistently updating the same message ID repeatedly, this can tie up many of the service’s db connections (if using a connection pool driver approach) as they each compete for and attempt to lock the record. If there is a non-trivial latency between the service nodes and the DB, this can cause a significant performance degradation and appear as very high response latency. In addition, if the service is responsible for other writes to the DB, this will affect those queries as well (assuming a single main RW db node).

Solution SYNC2 (Generic SQL implementation)

This solution is very similar to SYNC1, in the way that it does not modify the existing architecture and makes virtually no changes to the database (except for removing the now unnecessary INSERT/UPDATE timestamp trigger).

We can achieve similar behavior to the conditional update in SYNC1, however we can now remove any MySQL specific code, and use generic SQL by wrapping our INSERT/UPDATE logic in a “SELECT … FOR UPDATE” transaction. The transaction (and any pseudocode) is listed below.

```
START TRANSACTION;

select *
from test1.events as e
where e.ID = @id and e.timestamp < @timestamp
FOR UPDATE;

# if more recent value does not exist, insert/update value

INSERT INTO test1.events(id,data,enable,timestamp)
VALUES (@id, @data, @enable, @timestmap)
ON DUPLICATE KEY UPDATE data=@data,enable=@enable, timestamp=@timestamp;

# if more recent value exists (first part of transaction returns 1 record), no-op

COMMIT;
```

This approach allows to essentially place a lock on a row for an id (primary key) that does not yet exist. That allows for the same kind of conditional logic (this time performed at the service-side, not on the DB) that determines whether we need to INSERT, UPDATE or NO-OP.

Trade-offs & Limitations

This allows us to apply the logic from SYNC1 to any rdbms (not limited to MySQL), however this approach is arguably less performant than SYNC1 since the query results must return to the service before the final part of the query is executed (and transaction is committed). Depending on the latency between the service and the DB, this can be a significant amount of time that the lock, as now the minimum amount of time to perform this query is 2 x RTT(service-to-db); not counting any connection establishment time, locking contention on the DB, or prepared-statement overhead.