SUBMISSION- ROUND 3

File Name: Team_PromptWizard_Round3.pdf
File URL:
https://docs.google.com/document/d/13dRUWPrbBmaPQJwJREohazEinHukyf9p1msS21LluK8/edit?usp=sharing

TEAM NAME: Prompt Wizzard

Member 1 NAME: Mohsin Syed

Member 2 NAME: Ammaar Ahmad Khan

TOOL USED: Claude Sonnet 4.5

PROBLEM-1:Message Explosion in a Global Chat Platform

Prompt 1:You are a Specialized Info of to make chat platforms . The context is that a chat application is deployed globally. each region has its own clusters to local databases for reduced latency during peak hours many messages are created on different servers at same time . Queries like fetch and return in same order. The Constains incude No centarlized counter network latency varies and clocks anre not in sync .My question is that how reliably can it order messages at scale and tell its structure in consice manner to a student

AI Response: **Distributed Message Ordering at Scale**

The Challenge

You need to order messages created across multiple servers globally where:

- No centralized counter available
- Clocks drift (can't trust timestamps alone)
- Network delays vary unpredictably

Solution: Hybrid Vector Clock + Lamport Timestamp

Structure

Message {

id: UUID

content: string

timestamp: {

wallClock: ISO8601 // Physical time (approximate)

logicalClock: long // Lamport counter

vectorClock: {nodeId: counter} // Causality tracker

}

nodeId: string // Which server created it

sequenceNum: long // Per-node counter

}

How It Works

1. Local Ordering (Per Server)

- Each server maintains its own monotonic counter
- Messages get: `(nodeId, sequenceNum)` pair
- Example: `(server-us-1, 1045)`

2. Causal Ordering (Between Servers)

- Use **Lamport timestamps**: increment counter on every event
- When server receives message: `localCounter = max(localCounter, receivedCounter) + 1`
- If A happened-before B, then timestamp(A) < timestamp(B)

3. Tie-Breaking

- When Lamport timestamps equal, use `nodeId` lexicographically
- Example: Both timestamp 1000 → `server-asia-2` < `server-us-1`

Database Schema

messages (

id UUID PRIMARY KEY,

node_id VARCHAR,

sequence_num BIGINT,

lamport_ts BIGINT,

```
  wall_clock TIMESTAMP,
  content TEXT,
  INDEX (lamport_ts, node_id)  -- For ordering queries
)
```

Fetching in Order

```
# Query with consistent ordering
SELECT * FROM messages
ORDER BY lamport_ts ASC, node_id ASC
LIMIT 100
```

Why This Works at Scale

✅ **No coordination needed** - each server acts independently
✅ **Causality preserved** - if message B replies to A, B always ordered after A
✅ **Deterministic** - same input always produces same order
✅ **Partition-tolerant** - works even with network splits

Trade-off

Messages created simultaneously on different servers may appear in arbitrary (but consistent) order. This is unavoidable in distributed systems without coordination.

PROBLEM-2: Ticket Booking System and Overselling Seats

Prompt 1: **Role**: You are an expert in product manager, computer systems, and software development. **Situation** An online booking system sells tickets for high-demand events. 1000s of users attempt to book the same seat at the exact same moment - initial implementation: read available seats, lock row, deduct seat, commit transaction - under load: deadlocks increase, latency spikes, throughput collapses team replaces this approach and suddenly: -overselling stops -throughput increases -locks almost disappear **Question**: what fundamental change in concurrency control strategy likely solved this problem **Output:**: explain what was the thing(s) and approach that actually had the best chance or reason of solving this problem. Keep your answer concise but explain clearly in explained points

AI Response:

The fundamental change was likely **shifting from pessimistic locking to optimistic concurrency control**, specifically using **compare-and-swap (CAS) operations** or **version-based updates**. Here's why this solved the problem:

Key Changes

1. Eliminated Row-Level Locks

- Original: Acquired exclusive locks on seat rows → caused lock contention with 1000s of concurrent users
- New: No locks held during read phase → eliminated deadlocks and lock wait times

2. Atomic Compare-and-Swap at Commit

- Used database mechanisms like:
    - `UPDATE seats SET status='booked', version=version+1 WHERE seat_id=X AND version=Y AND status='available'`
    - Or PostgreSQL's `SELECT FOR UPDATE SKIP LOCKED`
- Only one transaction succeeds when multiple attempt the same seat simultaneously
- Losers fail fast and retry with a different seat

3. Version/Timestamp Checking

- Each seat record has a version number or timestamp
- Transaction reads seat + version, then commits only if version unchanged
- Detects conflicts without holding locks during user think-time

Why This Works Better

- **No deadlocks**: No lock acquisition ordering issues
- **Higher throughput**: Reads don't block writes; failed transactions detected instantly
- **No overselling**: Atomic CAS ensures exactly one winner per seat
- **Better scalability**: Lock-free reads allow massive read concurrency

The core insight: **detect conflicts at commit time rather than prevent them with locks** - perfect for high-contention scenarios where conflicts are common but each transaction touches only one seat.