Project: Concert Ticket Reservation System

Narawit Panyapunyarat

9 Nov 2025

Executive Summary

Overview

This document addresses performance and concurrency challenges in the Concert Ticket Reservation System affecting user experience and data integrity.

Key Challenges

- **Performance Degradation:** Slow response times (3-5s) under high traffic
- **Race Condition:** Multiple users can book the same seat simultaneously

Proposed Solutions

- **Performance:** Multi-layered optimization (frontend, backend, infrastructure)
- **Concurrency:** Four approaches evaluated from simple to advanced

Recommendation

Implement Pessimistic Locking with Database Transactions

- Eliminates 100% of race conditions
- Handles current traffic effectively
- Easy to maintain

Question 1:
Website Performance Optimization

How to Optimize Website Performance with Intensive Data and High Traffic

Frontend Optimization

Code Splitting & Lazy Loading

- Load components only when needed using dynamic imports
- Reduce initial bundle size with route-based splitting
- Improves First Contentful Paint (FCP) by 40-60%

Image Optimization

- Use Next.js Image component for automatic optimization
- Serve modern formats (WebP, AVIF) with fallbacks
- Implement responsive images for different screen sizes

Pagination & Virtual Scrolling

- Don't load all data at once - implement pagination (20-50 items per page)
- Use infinite scroll for better UX
- Virtual scrolling for long lists (render only visible items)

Client-side Caching

- React Query or SWR for smart data fetching
- Stale-while-revalidate strategy
- Reduces unnecessary API calls by 70-80%

Backend Optimization:

Database Indexing

- Index frequently queried fields: userId, concertId, createdAt
- Composite indexes for complex queries
- Can reduce query time by 80-90%

Query Optimization

- SELECT only needed fields, not entire records
- Implement pagination at database level
- Use proper JOINs to avoid N+1 problems

Caching Layer (Redis)

- Cache concert lists, available seats, user sessions
- Set appropriate TTL (5-60 minutes depending on data type)
- Reduces database load by 60-70%

Connection Pooling

- Configure Prisma connection pooling
- Prevents connection exhaustion
- Handles 5-10x more concurrent requests

Infrastructure Optimization

Content Delivery Network (CDN)

- Serve static assets through CloudFlare/AWS CloudFront
- Reduces latency by 50-70% for global users

Load Balancing

- Deploy multiple backend instances
- Distribute traffic with Nginx or AWS ALB
- Enables horizontal scaling

Database Replication

- Read replicas for SELECT queries
- Master for write operations
- Distributes load and improves availability

Question 2:
Concurrent Reservation Handling

How to Handle Concurrent Ticket Reservations

The Problem

Current implementation has a race condition - multiple users can reserve the same seat simultaneously because checking availability and creating reservations aren't atomic operations.

Solution 1: Optimistic Locking

Approach: Add version field to Concert model; increment on each update. Update only succeeds if version matches.
Pros: Simple, no locks, low overhead
Cons: Requires retry logic, poor UX during high traffic
Best for: Low contention scenarios

Solution 2: Pessimistic Locking with Transactions

Approach: Use database row locking (FOR UPDATE) within transactions to ensure exclusive access during reservation.
Pros: Guaranteed consistency, no retries needed, built-in Prisma support
Cons: Can create bottlenecks with very high concurrency
Best for: Critical operations requiring strong consistency

Solution 3: Redis Distributed Lock

Approach: Use Redis with Redlock algorithm for distributed locking. Atomically decrement seat counter in Redis, then persist to database.
Pros: Extremely fast, highly scalable, works across multiple servers
Cons: Requires Redis infrastructure, additional complexity
Best for: High concurrency, horizontal scaling

Solution 4: Redis + Ably Realtime

Approach: Combine Redis locking with Ably pub/sub for real-time seat availability updates to all connected clients.
How it works:

- Backend uses Redis lock for reservations
- Publishes events to Ably channel on each change
- All clients subscribe and receive instant updates
- UI shows live seat countdown

Pros: Best UX, instant updates, handles massive concurrency, excellent for marketing
Cons: Highest complexity and cost, requires three systems (PostgreSQL, Redis, Ably)
Best for: Premium events, flash sales, 10,000+ concurrent users