Transactions Management in High Performance System

Author: Kamil Rafikov
Email: mailbox@kamil-rafik.com
WhatsApp: +996 755 439 777
Telegram: kamilrafikov

Updated at January 18th, 2026.
Edition: 1 (final fact-checking not yet performed).

Table of Contents

Introduction

I have limited practical experience in the reviewed topic. You should check all provided information in other sources before applying it to your project.

UI/UX layer

The following techniques may be applied on UI/UX layer for managing transactional procedures:

a) locking UI for the timespan of transaction processing; I have observed locked UI with processing latencies up to 15-30 seconds as in a world known booking management application (while booking cheap hotels in provincial towns of developing countries where definitely no competition for rooms is present) as in a popular money transfer service between ex-USSR countries; probably, typical users are accustomed to such behavior in different applications, and that's why such approach may be considered as acceptable one in some circumstances; you should understand that such approach only blocks identical (or just multiple) requests from the same user and decreases load on the system, no more;

b) telling user that transaction is successful but needs to be *"confirmed"*; then showing successful confirmation notification with a delay up to several minutes; I have observed such approach in a world known booking application too; it is not really clear whether it is application delay or an issue on counterparty (hotel) side; considering that I never received failed confirmation

notification on several tens of bookings, I guess it is a technique to give the system additional time for processing; probably, typical users are accustomed to such behavior in different applications, and that's why such approach may be considered as acceptable one in some circumstances too.

Even if all aforementioned options look like synchrounous and interface-locking for user, they should be implemented in asynchrounous way in code to avoid real locks on frontend code level.

In all aforementioned options an animated counter of remaining time may be shown to display fake or real remaining time until the completion of transaction.

All aforementioned options should be combined with detailed logging and monitoring on server-side to track any hidden issues. Ideally, they should be used as last-resort options only.

Using fake statuses or fake counters may be illegal in finances-related business domains. You should consult a lawyer to be sure.

Mandatory idempotency keys must be implemented to avoid duplication of requests due to technical issues. For a case of request declined due to idempotency key matching to processed/completed transactions, UI/UX should not display failure to users but instead of it a status of found transaction must be returned.

Infrastructure layer

Segmentation of project audience by geographical or business logic rules with redirecting requests to appropriate nodes should be performed. Nodes must be placed physically as close as possible to user locations. Partitioning and/or sharding of data should be performed accordingly.

In such case data consistency may be reached only upon some latency, it should be checked whether business rules allow this.

Business rules logic code layer

The following techniques may be applied on business rules logic code layer for managing transactional procedures:

a) implementing lightweight services for processing critical transactional operations;

b) processing of idempotency keys received from client side is mandatory; idempotency keys must be cached inside in-memory storage;

c) enabling locks by data ID, or user ID, or both with an in-memory storage until the processing is completed to prohibit user from sending additional requests consciously.

If request is passed to scheduled job then all aforementioned techniques may be implemented on job level too.

Data operations logic code layer

The following techniques may be applied on data layer for managing transactional procedures.

Basic solutions:

a) avoiding to aquire explicit locks on data;

b) if data locks are used, then implementing global standard locking order (by sorted record IDs) to avoid the most obvious deadlocks; avoiding usage of some foreign keys and some triggers to eliminate risk of deadlocks that are hardly detectable in code reviews;

c) if data locks are used, then catching deadlock termination errors and lock timeout termination errors separately from other errors, and retrying operations inside the same request (or job) for acceptable number of times within timespan acceptable for UI/UX (with checking total system load to avoid collapse);

d) many modern relational DBMS can handle rather large amount of concurrent operations; that's why optimistic concurrency control (OCC) may be enough (exact benchmarks for each business logic case and each DBMS should be searched online); updated record should have version number (or some another criteria, like balance higher than subtracted amount) that is checked and updated in UPDATE clause; then number of updated rows is checked; such approach is ideal for read-heavy scenarios with minimal risks of conflicts; also, such approach may be implemented inside internals of some DBMS already, so you may not need to implement it in code.

Complex solutions:

a) collecting operations requests from multiple sources into batches and apply to the main record a combined result from every batch;

b) if inconsistencies (e.g. negative balances) are acceptable in some amount, then allowing them with subsequent resolution on user level or within some timespan;

c) implementing single ordered write operations sequence (e.g. with events management system) for logically partitioned data that is processed by single consumer service; it works fine in simple scenarios, but if multiple records must be updated transactionally then you may need also one or both of the following approaches;

- most common combinations of records (e.g. balances of related persons or business entities) should be partititioned and assigned to consumer together, with aggregation of IDs;
- perform update of each record with keeping history of changes, marking unclear status of record, and restoring data on failure in each related operation; intermediate results in multi-record operations will be visible and may cause issues as on user-level as on business rules logic level;

d) implementing storage of operations on values that cause the most part of concurrency issues instead of modifying the values themselves, and perform recalculations with a latency acceptable by business rules;

e) if updates of single dataset are coming in large amounts from multiple sources but for each source all updates are sequential then each source may operate with its own version of dataset, and merges may occur upon some latency acceptable by business rules; resolution of inconsistencies may be performed as on user-level as with some consensus algorithm.

DBMS layer

The following techniques may be applied on DBMS layer for managing transactional procedures:

a) relaxing transaction isolation level (if issues appearing in results are acceptable by business rules);

b) keeping separate copies of data for reading and writing if business rules accept synchronization with long latencies;

c) keeping all data in memory storage with flushing to hard drive in low load hours; in such scenario crash recovery issues must be reviewed with your technical operations team.

Summary

I hope this document demonstrates my capability to design data management in high performance systems, I hope you enjoyed reading, and I expect long-term productive work with you. Thank you for spending your time!