# A beginner's guide to ACID and database transactions

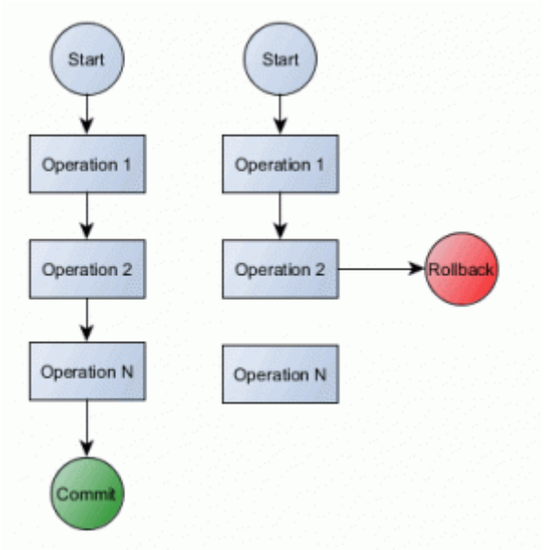
JANUARY 5, 2014 / VLADMIHALCEA

*(Last Updated On: January 29, 2018)*

# Introduction

Transactions are omnipresent in today's enterprise systems, providing data integrity even in highly concurrent environments. So let's get started by first defining the term and the context where you might usually employ it.

A transaction is a collection of read/write operations succeeding only if all contained operations succeed.




Inherently a transaction is characterized by four properties (commonly referred as ACID):

1. Atomicity
2. Consistency
3. Isolation
4. Durability

In a relational database, every SQL statement must execute in the scope of a transaction. Without defining the transaction boundaries explicitly, the database is going to use an implicit transaction which is wraps around every individual statement. The implicit transaction begins before the statement is executed and end (commit or rollback) after the statement is executed.
The implicit transaction mode is commonly known as autocommit.

For an enterprise application, the auto-commit mode is something you'd generally want to avoid since it has serious performance penalties, and it doesn't allow you to include multiple DML operations in a single atomic Unit of Work.

It's very important to understand those, hence we will discuss each and every one of them as follows.

# Atomicity

Atomicity takes individual operations and turns them into an all-or-nothing unit of work, succeeding if and only if all contained operations succeed.

A transaction might encapsulate a state change (unless it is a read-only one). A transaction must always leave the system in a consistent state, no matter how many concurrent transactions are interleaved at any given time.

# Consistency

Consistency means that constraints are enforced for every committed transaction. That implies that all Keys, Data types, Checks and Trigger are successful and no constraint violation is triggered.

# Isolation

Transactions require concurrency control mechanisms, and they guarantee correctness even when being interleaved. Isolation brings us the benefit of hiding uncommitted state changes from the outside world, as failing transactions shouldn't ever corrupt the state of the system. Isolation is achieved through concurrency control using pessimistic or optimistic locking mechanisms.

# Durability

A successful transaction must permanently change the state of a system, and before ending it, the state changes are recorded in a persisted transaction log. If our system is suddenly affected by a system crash or a power outage, then all unfinished committed transactions may be

replayed.

For messaging systems like JMS, transactions are not mandatory. That's why we have non-transacted acknowledgement modes.

File system operations are usually non-managed, but if your business requirements demand transaction file operations, you might make use a tool such as XADisk.

While messaging and file systems use transactions optionally, for database management systems transactions are compulsory.

# Challanges

ACID is old school. Jim Gray described atomicity, consistency and durability long before I was even born. But that particular paper doesn't mention anything about isolation. This is understandable if we think of the production systems of the late 70's, which according to Jim Gray:

> "At present, the largest airlines and banks have about 10,000 terminals and about 100 active transactions at any instant".

So all efforts were spent on delivering correctness rather than concurrency. Things have changed drastically ever since, and nowadays even modest set-ups are able to run 1000 TPS.

From a database perspective, the atomicity is a fixed property, but everything else may be traded off for performance/scalability reasons.

If the database system is composed of multiple nodes, then distributed system consistency (C in CAP Theorem not C in ACID) mandates that all changes be propagated to all nodes (multi-master replication). If slaves nodes are updated asynchronously then we break the consistency rule, the system becoming "eventually consistent".

> Peter Bailis has a very good article explaining the difference between Consistency in CAP Theorem and Consistency in ACID.

A transaction is a data state transition, so the system must operate as if all transactions occur in a serial form even if those are concurrently executed.
If there would be only one connection running at all times, then serializability wouldn't impose any concurrency control cost. In reality, all transactional systems must accommodate concurrent requests, hence serialization has its toll on scalability. The Amdahl's law describes the relation between serial execution and concurrency:

> "The speedup of a program using multiple processors in parallel computing is limited by the time needed for the sequential fraction of the program."

As you'll see later, most database management systems choose (by default) to relax correctness guarantees to achieve better concurrency.

Playing with durability makes sense for highly performing clustered databases if the enterprise system business requirements don't mandate durable transactions. But, most often durability is better off untouched.

# Isolation Levels

Although some database management systems offer MVCC, usually concurrency control is achieved through locking. But as we all know, locking increases the serializable portion of the executed code, affecting parallelization.

The SQL standard defines four Isolation levels:

- READ_UNCOMMITTED
- READ_COMMITTED
- REPEATABLE_READ
- SERIALIZABLE

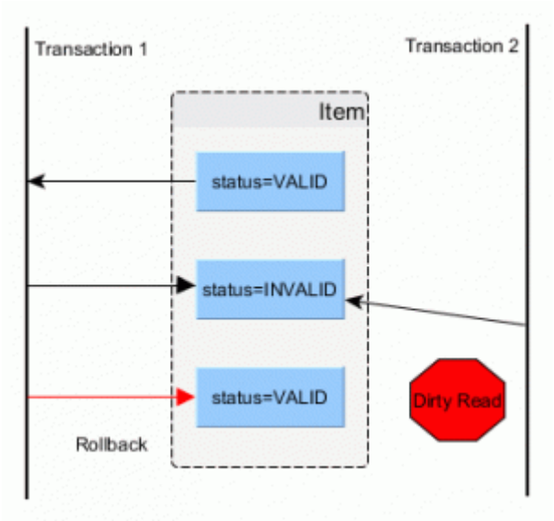
All but the SERIALIZABLE level are subject to data anomalies (phenomena) that might occur according to the following pattern:

| Isolation Level | Dirty read | Non-repeatable read | Phantom read |
|---|---|---|---|
| READ_UNCOMMITTED | allowed | allowed | allowed |
| READ_COMMITTED | prevented | allowed | allowed |
| REPEATABLE_READ | prevented | prevented | allowed |
| SERIALIZABLE | prevented | prevented | prevented |

# Phenomena

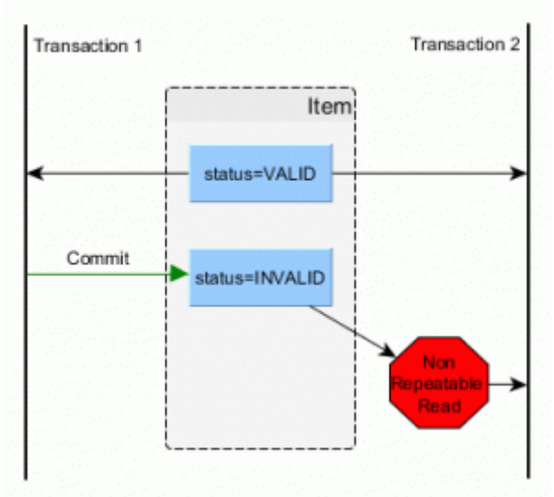
But what are all those phenomena we have just listed? Let's discuss each and every one of them.

## Dirty read



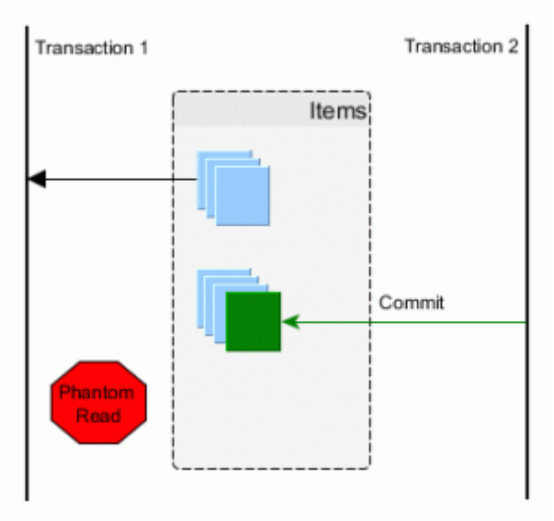

A dirty read happens when a transaction is allowed to read uncommitted changes of some other running transaction. This happens because there is no locking preventing it. In the picture above, you can see that the second transaction uses an inconsistent value as of the first transaction had rollbacked.

## Non-repeatable read




A non-repeatable read manifests when consecutive reads yield different results due to a concurring transaction that has just updated the record we're reading. This is undesirable since we end up using stale data. This is prevented by holding a shared lock (read lock) on the read record for the whole duration of the current transaction.

## Phantom read




A phantom read happens when a second transaction inserts a row that matches a previous select criteria of the first transaction. We, therefore, end up using stale data, which might affect our business operation. This is prevented using range locks or predicate locking.

## Even more phenomena

Even if not mentioned in the SQL Stanard, there are even more phenomena that you should be aware of, like:

- Lost Updates
- Read Skew
- Write Skew

Knowing when these phenomena can occur can addressing them properly is what data integrity is all about.


# Default Isolation Levels

Even if the SOL standard mandates the use of the SERIALIZABLE isolation level, most database management systems use a different default level.

| Database | Default isolation Level |
| --- | --- |
| Oracle | READ_COMMITTED |
| MySQL | REPEATABLE_READ |
| Microsoft SQL Server | READ_COMMITTED |
| PostgreSQL | READ_COMMITTED |
| DB2 | CURSOR STABILITY |

Usually, READ_COMMITED is the right choice, since not even SERIALIZABLE can protect you from a lost update where the reads/writes happen in different transactions (and web requests). You should take into consideration your enterprise system requirements and set up tests for deciding which isolation level best suits your needs.

**Related**

How to prevent lost updates in long conversations
September 22, 2014
In "Hibernate"

A beginner's guide to database locking and the lost update phenomena
September 14, 2014
In "Hibernate"

A beginner's guide to Dirty Read anomaly
May 23, 2018
In "Database"

Categories: Transactions  Tags: ACID, Database, isolation levels, transactions

← Asciidoctor for collaborative book writing

MongoDB time series: Introducing the aggregation framework →

## Leave a Reply

Your email address will not be published. Required fields are marked *

**Comment**

**Name** *

**Email** *

Website

Save my name, email, and website in this browser for the next time I comment.

Sign me up for the newsletter!

Post Comment

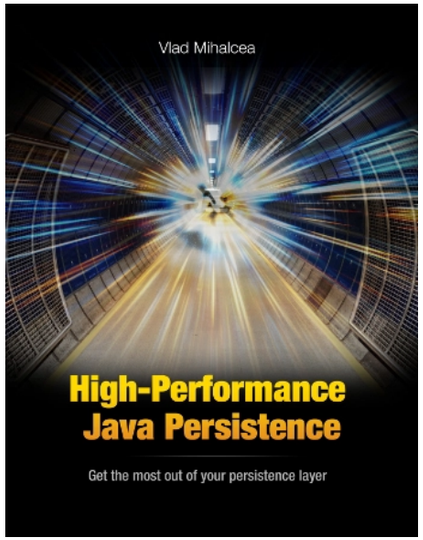
Notify me of follow-up comments by email.

This site uses Akismet to reduce spam. Learn how your comment data is processed.

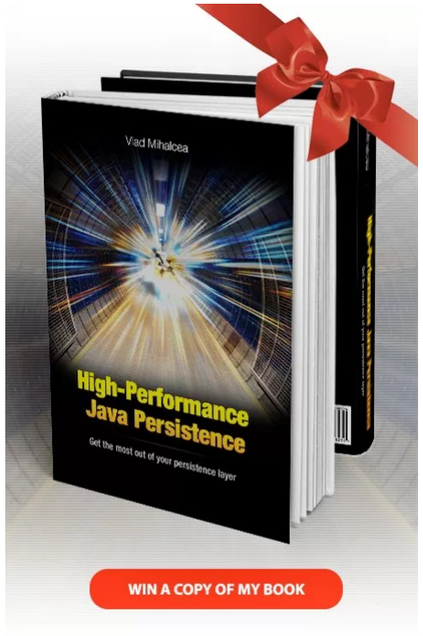
JDK.IO WORKSHOP – COPENHAGEN

VIDEO COURSE


HIGH-PERFORMANCE JAVA PERSISTENCE




WIN A COPY OF MY BOOK!



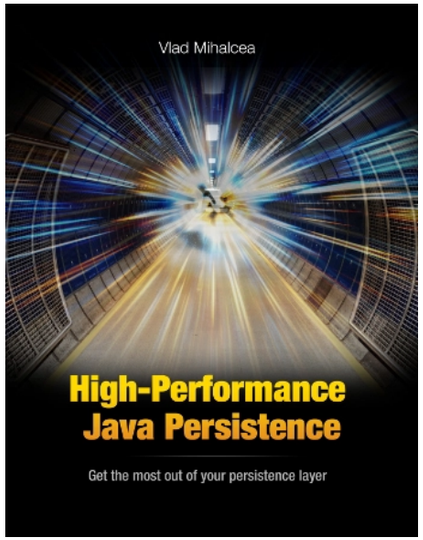


# HIBERNATE PERFORMANCE TUNING TIPS

HYPERSISTENCE

## WIN A COPY OF MY BOOK!


## HIGH-PERFORMANCE JAVA PERSISTENCE




VIDEO COURSE


JDK.IO WORKSHOP – COPENHAGEN

JDKIO
0xCAFEBABE
CONFERENCE
2018

**WORKSHOP**

**High-Performance Java Persistence**

RSS - Posts

RSS - Comments

ABOUT

About

Privacy Policy

Terms of Service