Optimistic vs. Pessimistic Locking

Last modified: Mar 24, 2021

Introduction

In this article, I'm going to explain what is the difference between optimistic and pessimistic locking, as well as when you should employ one or the other concurrency control strategies.

Conflicts

At the Networking course in college, I learned that there are two ways of dealing with conflicts or collisions:

- detect and retry, and that's exactly what Ethernet does

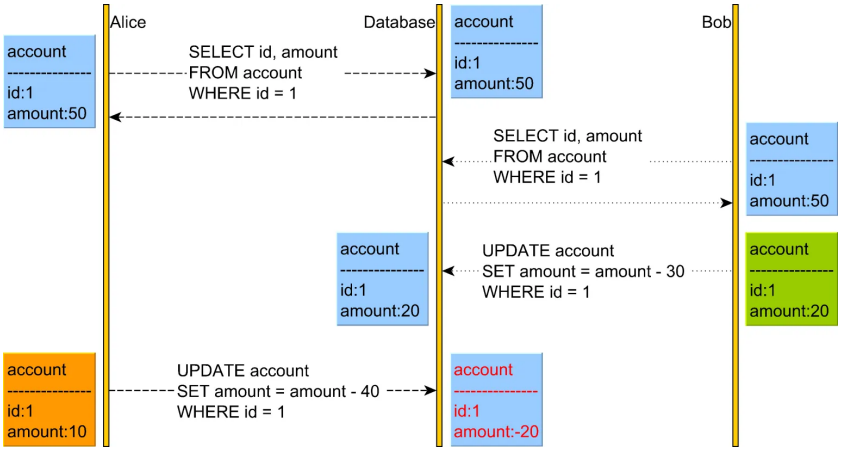
using a database system.

We could allow the conflict to occur, but then we need to detect it upon committing our transaction, and that's exactly how optimistic locking works.

If the cost of retrying is high, we could try to avoid the conflict altogether via locking, which is the principle behind how pessimistic locking works.

The Lost Update anomaly

Let's consider the Lost Update anomaly, which can happen on any database running under the Read Committed isolation level:



The diagram above illustrates the following situation:

- Alice reads the account balance, and the value is 50.

thinking that the final balance will be `10`.

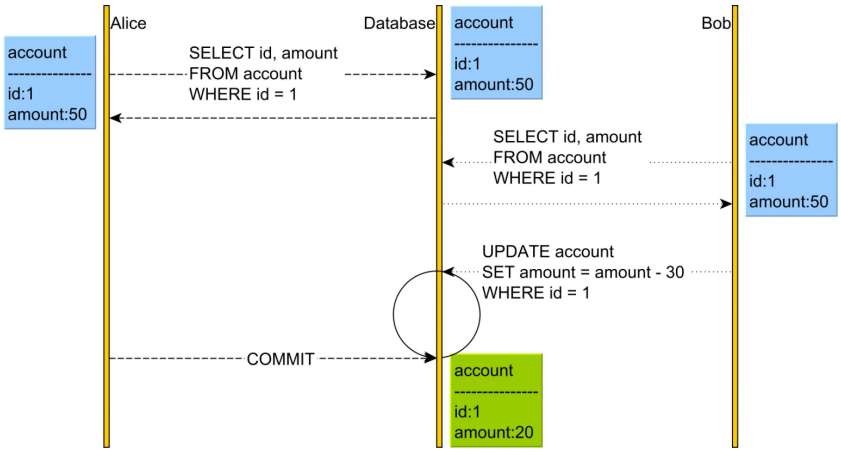
- However, since the valance has changed, Alice's UPDATE is going to leave the account balance in a negative value.

This transaction schedule is not Serializable because it's neither equivalent to Alice's reads and writes followed by Bob's read and writes or Bob executing his transaction first followed by Alice executing her transaction right after.

The reads and the writes are interleaves, and that's why the Lost Update anomaly is generated.

Pessimistic Locking

Pessimistic locking aims to avoid conflicts by using locking.



In the diagram above, both Alice and Bob will acquire a read (shared) lock on the `account` table row upon reading

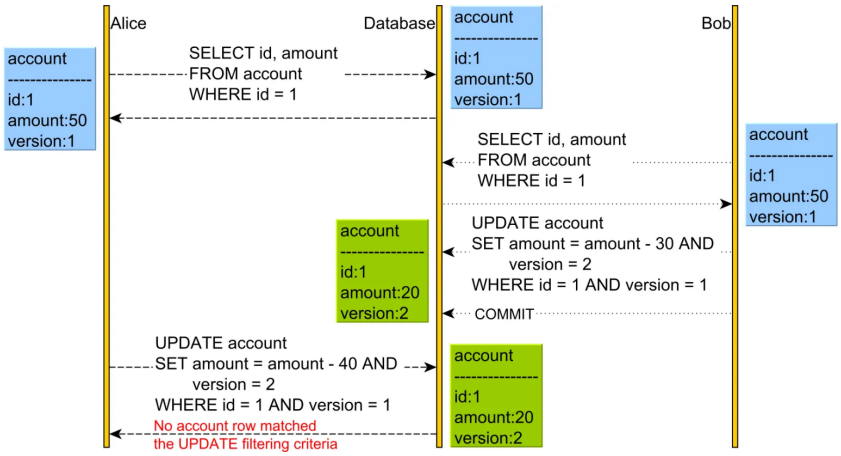
of them can change it until one releases the read lock they acquired. This is because a write operation requires a write (exclusive) lock acquisition, and read (shared) locks prevent write (exclusive) locks.

For this reason, Bob's UPDATE blocks until Alice releases the shared lock she has acquired previously.

> When using SQL Server, the database acquires the shared locks automatically when reading a record under Repeatable Read or Serializable isolation level because SQL Server uses the 2PL (Two-Phase Locking) algorithm by default.
>
> MySQL also uses pessimistic locking by default when using the Serializable isolation level and optimistic locking for the other less-strict isolation levels.

Optimistic Locking

Optimistic Locking allows a conflict to occur, but it needs to detect it at write time. This can be done using either a physical or a logical clock. However, since logical clocks are superior to physical clocks when it comes to implementing a concurrency control mechanism, we are going to use a `version` column to capture the read-time row snapshot information.



So, when reading the `account` record, both users read its current version. However, when Bob changes the `account` balance, he also changes the version from `1` to `2`.

Afterward, when Alice wants to change the `account` balance, her UPDATE statement will not match any record since the version column value is no longer `1`, but `2`.

Therefore, the `executeUpdate` method of the UPDATE `PreapredStatement` is going to return a value of `0`, meaning that no record was changed, and the underlying data access framework will throw an `OptimisticLockException` that will cause Alice's transaction to rollback.

So, the Lost Update is prevented by rolling back the subsequent transactions that are operating on state data.

use MVCC (Multi-Version Concurrency Control), which is based on optimistic locking.

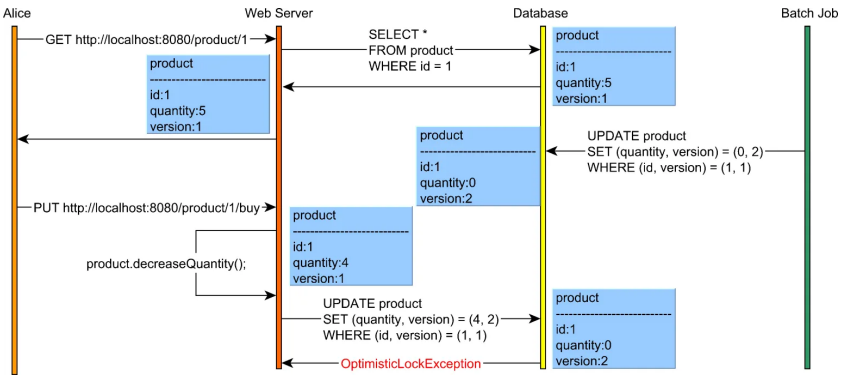
So, in MVCC, readers don't block writers and writers don't block readers, allowing conflicts to occur. However, at commit time, conflicts are detected by the transaction engine and the conflicting transactions are rolled back.

Application-level transactions

Relational database systems have emerged in the late '70s and early '80s when clients would connect to a mainframe via a terminal. However, nowadays, that's not the case when using a web browser.

So, we no longer execute reads and writes in the context of the same database transaction, and Serializability is no longer sufficient to prevent a Lost Update in a long conversation.

For instance, considering we have the following use case:



So, optimistic locking can help you prevent Lost Updates even when using application-level transactions that incorporate the user-think time as well.

of retrying a transaction is very high or when contention is so large that many transactions would end up rolling back if optimistic locking were used.

On the other hand, optimistic locking works even across multiple database transactions since it doesn't rely on locking physical records.

Related

In "Database"

Category: Database, Hibernate Tags: anomaly, lost updates, optimistic locking, pessimistic locking, Serializable

← JPA Default Fetch Plan SQL Server deadlock trace flags →

2 Comments on "Optimistic vs. Pessimistic Locking"

Raheem

August 30, 2021

Great article, thank you, sir!

Reply

vladmihalcea

August 30, 2021

Thanks.

Reply

Leave a Reply

Before posting the comment, please take the time to read the FAQ page

Name *

Email *

Website

Notify me of follow-up comments by email.

Post Comment

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