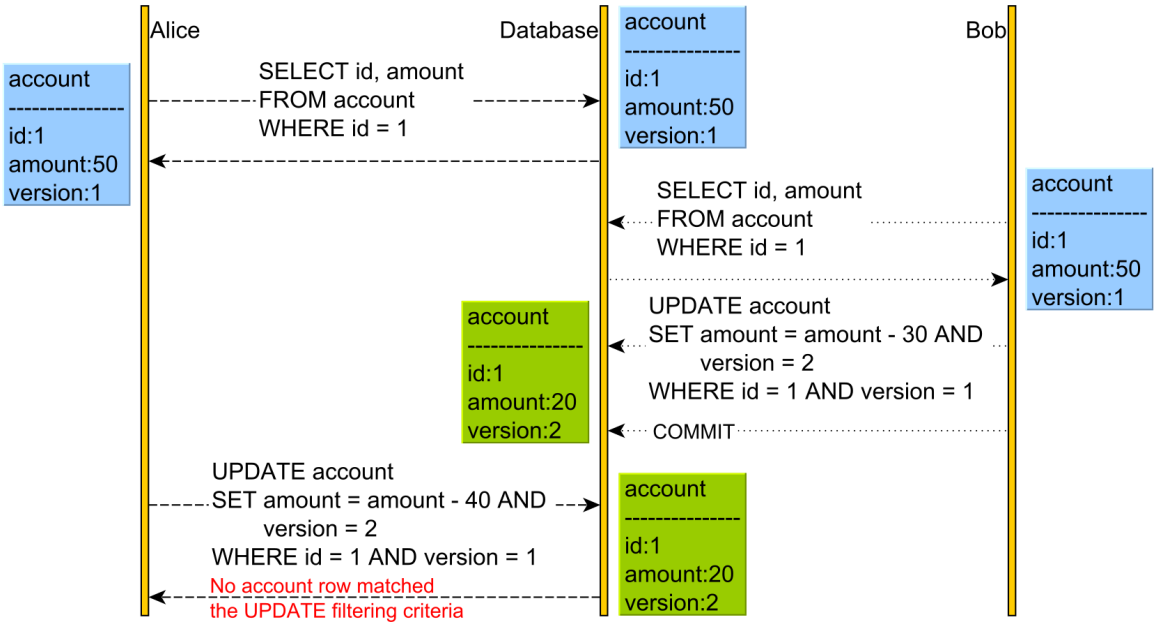
Optimistic vs. Pessimistic Locking

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.

The `version` column is going to be incremented every time an UPDATE or DELETE statement is executed while also being used for matching the expected row snapshot in the WHERE clause.



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`.

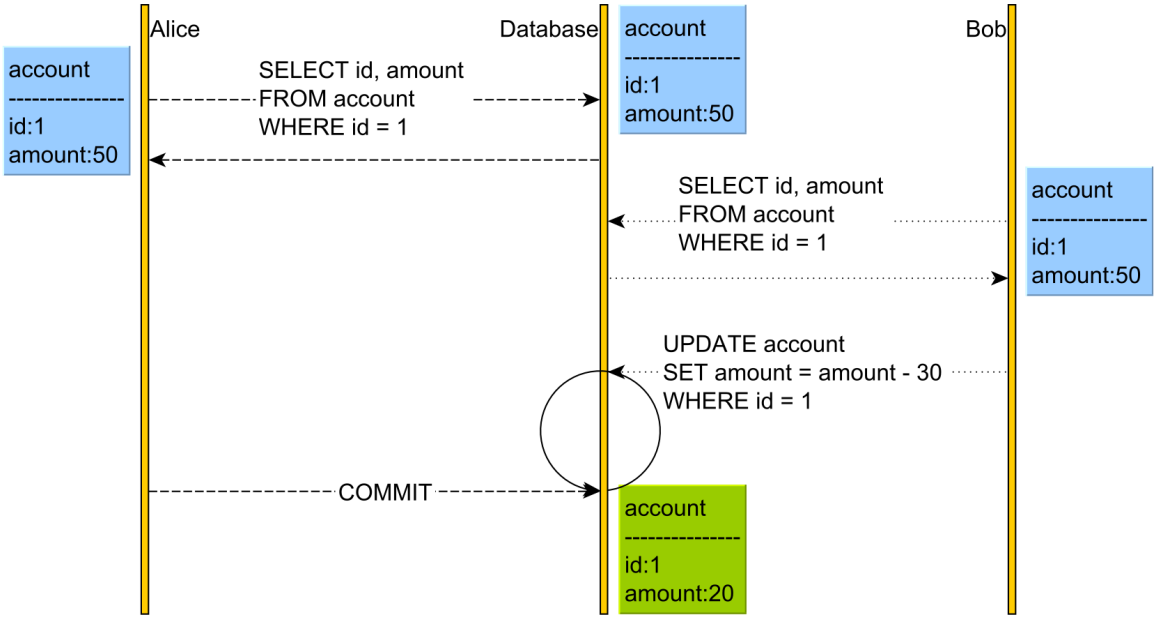
Afterward, when Alice wants to change the `account` balance, her UPDATE statement will not match any record since the version column value is no longer `1`, but `2`.

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.

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 it.

Because both Alice and Bob hold the read (shared) lock on the `account` record with the identifier value of `1`, neither 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.

DeadLock Solution

- Prevention (disallow one of the four requirements)
- Avoidance (study what is required by all before beginning)
- Detection (using time-outs or wait-for graphs) and recovery

Saga Design Pattern

Saga are series of local transactions, where each local transaction mutates and persist the entities along with some flag indicating the phase of the global transaction and commits the change. It guarantees data consistency in a distributed framework.

Two ways to implement:

- Events/Choreography Saga: no central coordination, each service produces and listens to the other service's events and decides if an action should be taken or not.
- Command/Orchestration Saga: a coordinator service is responsible for centralizing the saga's decision making and sequencing business logic