Vlad Mihalcea

Transactions and Concurrency Control

Get the most out of your persistence layer

Vlad Mihalcea

Transactions and Concurrency Control

Cluj-Napoca

2020

Publisher:

Vlad Mihalcea

Cluj-Napoca Romania

mihalcea.vlad@gmail.com

Cover design:

Dan Mihalcea

danmihalcea@gmail.com

Cover photo:

Carlos ZGZ[1] - CC0 1.0[2]

[1] https://www.flickr.com/photos/carloszgz/19980799311/

[2] https://creativecommons.org/publicdomain/zero/1.0/

This book is an excerpt from my High-Performance Java Persistence book.

If you enjoy reading the Transactions and Concurrency Control book, you will definitely love reading High-Performance Java Persistence and learn how to run your data access layer at high speed.

*With this coupon, you can buy High-Performance Java Persistence with a **10% discount**.*

Thanks for reading my book and stay tuned for more!`

Contents

1. Transactions

A database system must allow concurrent access to the underlying data. However, shared data means that read and write operations must be synchronized to ensure that data integrity is not compromised.

To control concurrent modifications, the Java programming language defines the `synchronized` keyword for two purposes:

- It can restrict access to a shared `Object` (to preserve invariants), so only a `Thread` can execute a routine at any given time.
- It propagates changes from the current `Thread` local memory to the global memory that is available to all running threads of executions.

This behavior is typical for other concurrent programming environments and database systems are no different. In a relational database, the mechanism for ensuring data integrity is implemented on top of transactions.

A transaction is a collection of read and write operations that can either succeed or fail together, as a unit. All database statements must execute within a transactional context, even when the database client does not explicitly define its boundaries.

In 1981, Jim Gray first defined the properties of a database transaction in his famous paper: The transaction concept: virtues and limitations[1]. Both this paper and the first versions of the SQL standard (SQL-86 and SQL-89) only used three properties for defining a database transaction: *Atomicity*, *Consistency*, and *Durability*.

Along with other relation database topics, the transaction research has continued ever since, and so the SQL-92 version introduced the concept of *Isolation Levels*. These four properties have been assembled in the well-known ACID (*Atomicity*, *Consistency*, *Isolation*, *and Durability*) acronym that soon became synonym with relation database transactions.

Knowing how database transactions work is very important for two main reasons:

- effective data access (data integrity should not be compromised when aiming for high-performance)
- efficient data access (reducing contention can minimize transaction response time which, in turn, increases throughput).

The next sections detail each transaction property in relation to high-performance data processing.

[1] http://research.microsoft.com/en-us/um/people/gray/papers/theTransactionConcept.pdf

1.1 Atomicity

Atomicity is the property of grouping multiple operations into an all-or-nothing unit of work, which can succeed only if all individual operations succeed. For this reason, the database must be able to roll back all actions associated with every executed statement.

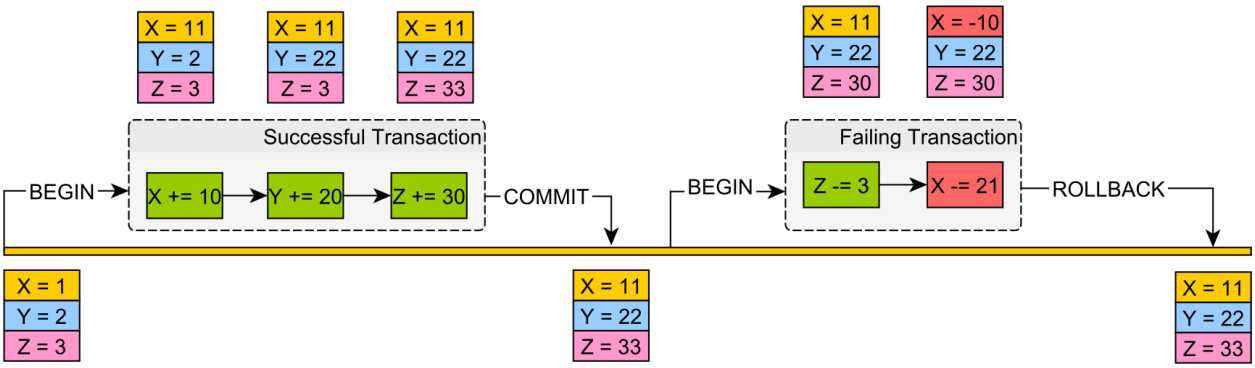

Figure 6.1: Atomic units of work

Write-write conflicts

Ideally, every transaction would have a completely isolated branch which could be easily discarded in case of a rollback. This scenario would be similar to how a *Version Control System* (e.g. *git*) implements branching. In case of conflicts, the Virtual Control System aborts the commit operation, and the client has to manually resolve the conflict. Unlike VCS tools, the relational database engine must manage conflicts without any human intervention.

For this reason, the database prevents write-write conflict situations, and only one transaction can write a record at any given time.

All statements are executed against the actual data structures (tables, indexes, in-memory buffers), only to be materialized at commit time. In case of rollback, the database must revert any pending changed datum to its previous state.

Oracle

The *undo tablespace*[a] stores the previous data versions in *undo segments*. Upon rolling back, the database engine searches the associated *undo segments* that can recreate the *before image* of every datum that was changed by the currently running transaction.

[a] https://docs.oracle.com/database/121/ADMIN/undo.htm#ADMIN11460

SQL Server

The *transaction log*[a] stores details about the currently running transactions and their associated modifications. The rollback process scans the *transaction log* backward to find the associated undo records. When the record is found, the database engine restores the before image of the affected datum.

To prevent the transaction log from filling up, the log must be truncated on a regular basis. Long-running transactions can delay the truncation process, so that is another reason to avoid them as much as possible.

[a] https://msdn.microsoft.com/en-us/library/ms190925.aspx

PostgreSQL

Unlike other database systems, PostgreSQL does not use a dedicated append-only undo log. Because of its multi-version nature, every database object maintains its own version history. In the absence of the log seek-up phase, the rollback process becomes much lighter as it only requires to switch from one version to the other.

The downside is that the previous version space is limited in size, and so it must be reused. The process of reclaiming the storage occupied by old versions is called VACUUMING.

Each transaction has an associated XID[a], and newer transactions must have a greater XID number than all previous ones.

The transaction XID is a 32-bit number so it can accommodate over 4 billion transactions. In a high-performance application, the transaction lifespan is very short, and if the VACUUM process is disabled, this threshold may be reached. When the XID counter reaches its maximum value, it wraps around and start again from zero.

The transactions issued prior to the XID wraparound have their identifiers greater than newer transactions started after the XID counter reset. This anomaly can cause the system to perceive older transactions as they were started in the future, which can lead to very serious data integrity issues.

[a] http://www.postgresql.org/docs/current/static/routine-vacuuming.html

MySQL

The *undo log* is stored in the *rollback segment*[a] of the system tablespace.

Each *undo log* is split into two sections, one responsible for rolling back purposes and the other for reconstructing the before image. The first section can be wiped out right after the transaction is ended, while the other needs to linger for as long as any currently running query or other concurrent transactions need to see a previous version of the records in question.

Behind the scenes, MySQL runs a *purge* process that cleans up the storage occupied by deleted records, and it also reclaims the undo log segments that are no longer required.

Long-running transactions delay the *purge* process execution, causing the undo log to grow very large, especially in write-heavy data access scenarios.

[a] https://dev.mysql.com/doc/refman/5.7/en/innodb-multi-versioning.html

1.2 Consistency

A modifying transaction can be seen as a state transformation, moving the database from one valid state to another. The relational database schema ensures that all primary modifications (insert/update/delete statements), as well as secondary ones (issued by triggers), obey certain rules on the underlying data structures:

- column types
- column length
- column nullability
- foreign key constraints
- unique key constraints
- custom *check* constraints.

Consistency is about validating the transaction state change so that all committed transactions leave the database in a proper state. If only one constraint gets violated, the entire transaction will be rolled back, and all modifications are going to be reverted.

Although the application must validate user input prior to crafting database statements, the application-level checks cannot span over other concurrent requests, possibly coming from different web servers. When the database is the primary integration point, the advantages of a strict schema become even more apparent.

MySQL

Traditionally, MySQL constraints are not strictly enforced[a], and the database engine replaces invalid values with predefined defaults:

- Out-of-range numeric values are set to either 0 or the maximum possible value.
- `String` values are trimmed to the maximum length.
- Incorrect Date/Time values are permitted (e.g. 2015-02-30).
- NOT NULL constraints are only enforced for single INSERT statements. For multi-row inserts, 0 replaces a null numeric value, and `''` is used for a null `String`.

Since the 5.0.2 version, *strict* constraints are possible if the database engine is configured to use a custom *sql mode*[b]:

```
SET GLOBAL sql_mode='POSTGRESQL,STRICT_ALL_TABLES';
```

Because the `sql_mode` resets on server startup, it is better to set it up in the MySQL configuration file:

```
[mysqld]
sql_mode = POSTGRESQL,STRICT_ALL_TABLES
```

[a] https://dev.mysql.com/doc/refman/5.7/en/constraint-invalid-data.html
[b] https://dev.mysql.com/doc/refman/5.7/en/sql-mode.html

Consistency as in CAP Theorem

According to the CAP theorem[a], when a distributed system encounters a network partition, the system must choose either *Consistency* (all changes are instantaneously applied to all nodes) or *Availability* (any node can accept a request), but not both. While in the definition of ACID, consistency is about obeying constraints, in the CAP theorem context, consistency refers to *linearizability*[b], which is an isolation guarantee instead.

[a] https://en.wikipedia.org/wiki/CAP_theorem
[b] http://www.bailis.org/blog/linearizability-versus-serializability/

1.3 Isolation

If there were only one user accessing the database, there would not be any risk of data conflicts. According to the Universal Scalability Law, if the sequential fraction of the data access patterns is less than 100%, the database system may benefit from parallelization.

By offering multiple concurrent connections, the transaction throughput can increase, and the database system can accommodate more traffic. However, parallelization imposes additional challenges as the database must interleave transactions in such a way that conflicts do not compromise data integrity. The execution order of all the currently running transaction operations is said to be *serializable* when its outcome is the same as if the underlying transactions were executed one after the other.

The serializable execution is, therefore, the only transaction isolation level that does not compromise data integrity while allowing a certain degree of parallelization. In 1981, Jim Gray described the largest airlines and banks as having 10 000 terminals and 100 active transactions, which explains why, up until SQL-92, serializable was the *de facto* transaction isolation level.

1.3.1 Concurrency control

To manage data conflicts, several concurrency control mechanisms have been developed throughout the years. There are two strategies for handling data collisions:

- Avoiding conflicts (e.g. *two-phase locking*) requires locking to control access to shared resources.
- Detecting conflicts (e.g. *Multi-Version Concurrency Control*) provides better concurrency, at the price of relaxing serializability and possibly accepting various data anomalies.

1.3.1.1 Two-phase locking

In 1976, Kapali Eswaran and Jim Gray (et al.) published The Notions of Consistency and Predicate Locks in a Database System[2] paper, which demonstrated that serializability could be obtained if all transactions used the *two-phase locking* (2PL) protocol.

Initially, all database systems employed 2PL for implementing serializable transactions, but, with time, many vendors have moved towards an MVCC (Multi-Version Concurrency Control) architecture. By default, SQL Server still uses locking for implementing the Serializability isolation level.

Because 2PL guarantees transaction serializability, it is very important to understand the price of maintaining strict data integrity on the overall application scalability and transaction performance.

[2] http://research.microsoft.com/en-us/um/people/gray/papers/On%20the%20Notions%20of%20Consistency%20and%20Predicate%20Locks%20in%20a%20Database%20System%20CACM.pdf

However, locking is not used only in 2PL implementations, and, to address both DML and DDL statement interaction and to minimize contention on shared resources, relational database systems use Multiple granularity locking[3].

Database objects are hierarchical in nature, a logical tablespace being mapped to multiple database files, which are built of data pages, each page containing multiple rows. For this reason, locks can be acquired on different database object types.

Locking on lower levels (e.g. rows) can offer better concurrency as it reduces the likelihood of contention. Because each lock takes resources, holding multiple lower-level locks can add up, so the database might decide to substitute multiple lower-level locks into a single upper-level one. This process is called *lock escalation*, and it trades off concurrency for database resources.

Each database system comes with its own lock hierarchy, but the most common types (even mentioned by the 2PL initial paper) remain the following ones:

- shared (read) lock, preventing a record from being written while allowing reads
- exclusive (write) lock, disallowing both read and write operations.

Locks alone are not sufficient for preventing conflicts. A concurrency control strategy must define how locks are being acquired and released because this also has an impact on transaction interleaving.

For this purpose, the 2PL protocol defines a lock management strategy for ensuring serializability. The 2PL protocol splits a transaction into two sections:

- expanding phase (locks are acquired, and no lock is released)
- shrinking phase (all locks are released, and no other lock is further acquired).

In lock-based concurrency control, all transactions must follow the 2PL protocol, as otherwise serializability might be compromised, resulting in data anomalies.

Transaction schedule

To provide recovery from failures, the transaction schedule (the sequence of all interleaved operations) must be strict. If a write operation, in a first transaction, happens-before a conflict occurring in a subsequent transaction, in order to achieve transaction *strictness*, the first transaction commit event must also happen before the conflict.

Because operations are properly ordered, strictness can prevent *cascading aborts (one transaction rollback triggering a chain of other transaction aborts, to preserve data consistency)*. Releasing all locks only after the transaction has ended (either commit or rollback) is a requirement for having a strict schedule.

[3] https://en.wikipedia.org/wiki/Multiple_granularity_locking

The following diagram shows how transaction interleaving is coordinated by 2PL:

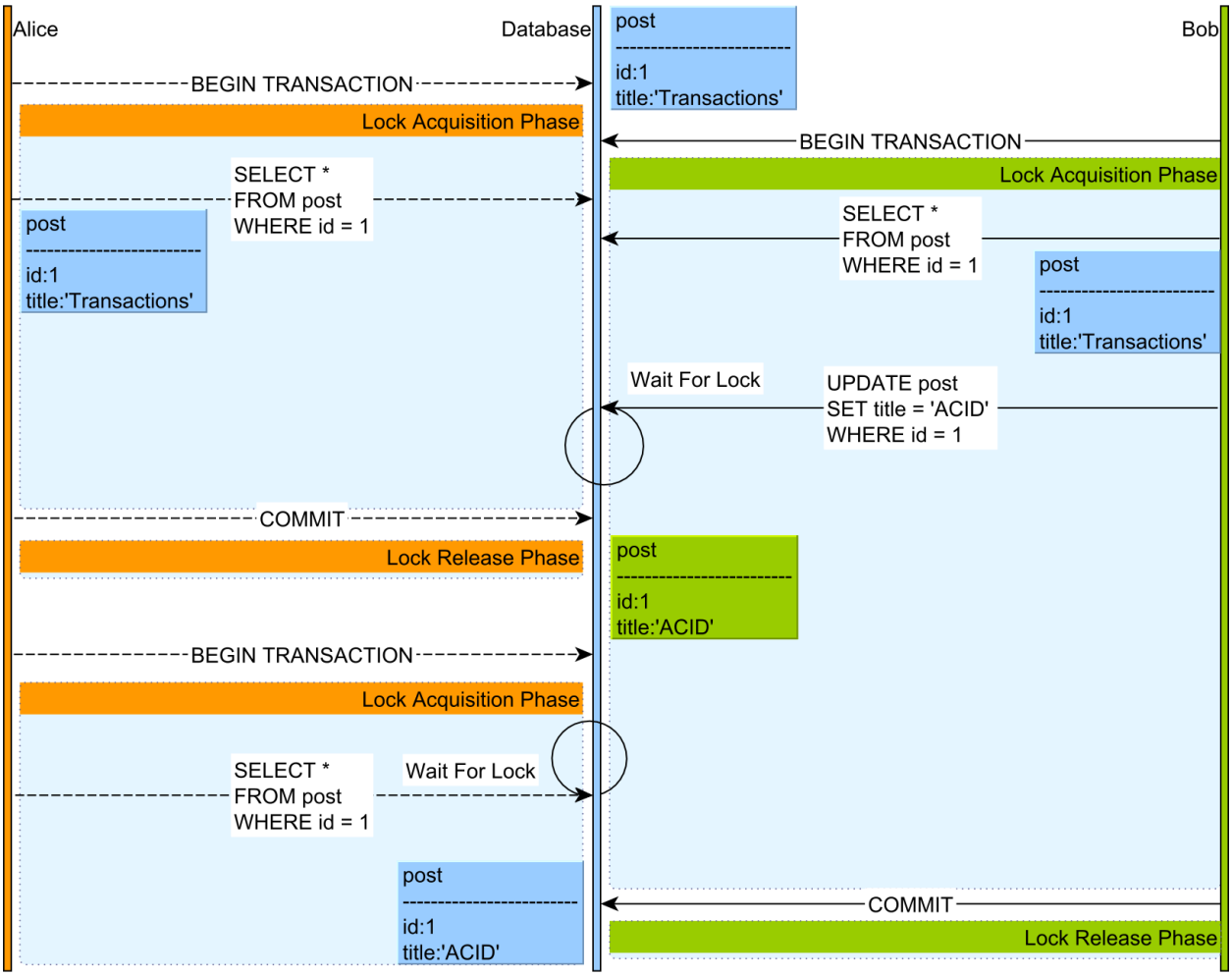

Figure 6.2: Two-phase locking

- Both *Alice* and *Bob* select a *post* record, both acquiring a shared lock on this record.
- When *Bob* attempts to update the *post* entry, his statement is blocked by the Lock Manager because *Alice* is still holding a shared lock on this database row.
- Only after *Alice*'s transaction ends and all locks are being released, *Bob* can resume his update operation.
- *Bob*'s update generates a lock upgrade, so the shared lock is replaced by an exclusive lock, which prevents any other concurrent read or write operation.
- *Alice* starts a new transaction and issues a select query for the same *post* entry, but the statement is blocked by the Lock Manager since *Bob* owns an exclusive lock on this record.
- After *Bob*'s transaction is committed, all locks are released, and *Alice*'s query can be resumed, so she gets the latest value of this database record.

Deadlocks

Using locking for controlling access to shared resources is prone to deadlocks, and the transaction scheduler alone cannot prevent their occurrences.

A deadlock happens when two concurrent transactions cannot make progress because each one waits for the other to release a lock. Because both transactions are in the lock acquisition phase, neither one releases a lock prior to acquiring the next one.

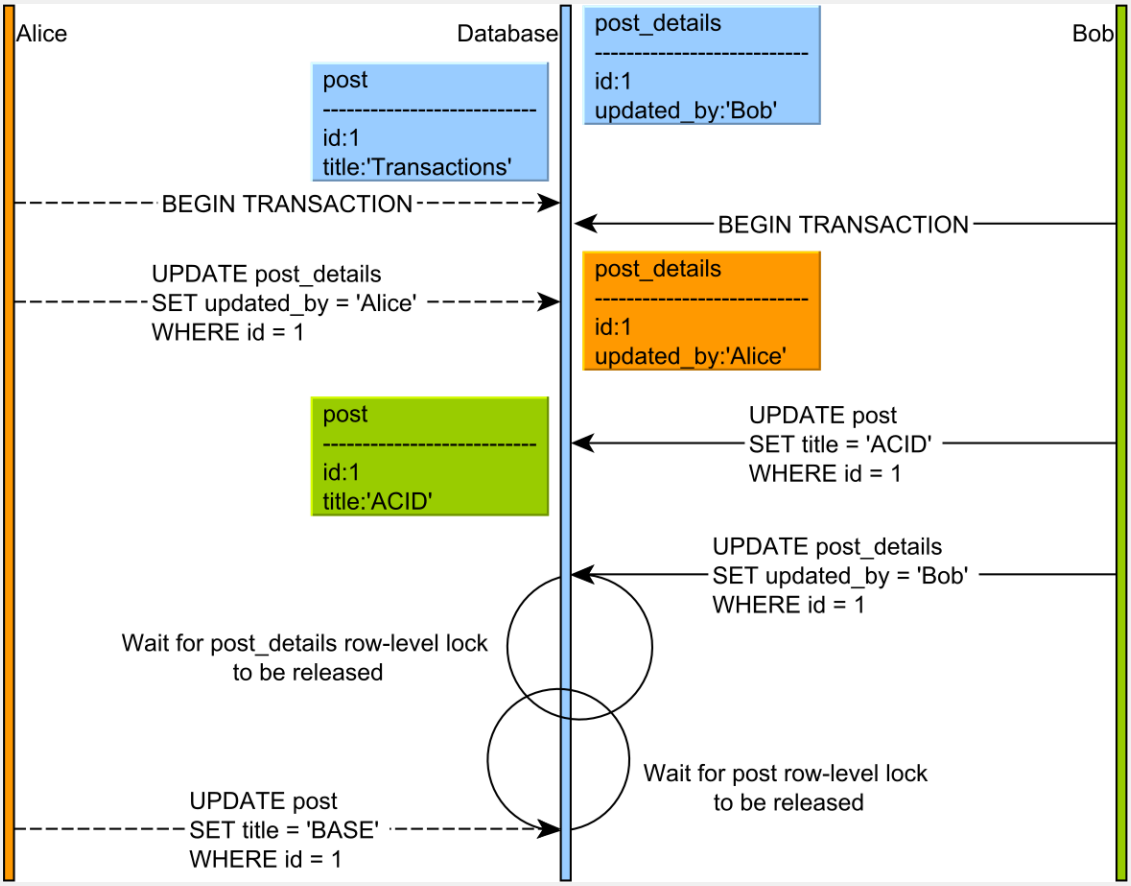

Figure 6.3: Dead lock

Preserving the lock order becomes the responsibility of the data access layer, and the database can only assist in recovering from a deadlock situation.

The database engine runs a separate process that scans the current conflict graph for lock-wait cycles (which are caused by deadlocks). When a cycle is detected, the database engine picks one transaction and aborts it, causing its locks to be released, so the other transaction can make progress.

1.3.1.2 Multi-Version Concurrency Control

Although locking can provide a serializable transaction schedule, the cost of lock contention can undermine both transaction response time and scalability. The response time can increase because transactions must wait for locks to be released, and long-running transactions can slow down the progress of other concurrent transactions as well. According to both Amdahl's Law and the Universal Scalability Law, concurrency is also affected by contention.

To address these shortcomings, the database vendors have opted for optimistic concurrency control mechanisms. If 2PL prevents conflicts, Multi-Version Concurrency Control (MVCC) uses a conflict detection strategy instead.

The promise of MVCC is that readers do not block writers and writers do not block readers. The only source of contention comes from writers blocking other concurrent writers, which otherwise would compromise transaction rollback and atomicity.

To prevent blocking, the database can rebuild previous versions of a database record so an uncommitted change can be hidden away from incoming concurrent readers. The lack of locking makes it more difficult to implement a serializable schedule, so the database engine must analyze the current interleaving operations and detect anomalies that would compromise serializability.

Oracle

Oracle does not implement 2PL at all, relying on MVCC mechanism for managing concurrent data access. Every query gets a point-in-time data snapshot and, depending on the isolation level, the timestamp reference can be relative to the current statement or to the current transaction start time.

To rebuild previous record versions, Oracle uses the undo segments[a], which already contain all the necessary data required for rolling back an uncommitted change. The point-in-time is based on the System Change Number (SCN), which is a logical timestamp reference and, unlike physical time, is guaranteed to be incremented monotonically.

Apart from MVCC, Oracle also supports explicit locking as well, using the `SELECT FOR UPDATE` SQL syntax.

[a] https://docs.oracle.com/database/121/CNCPT/consist.htm#CNCPT221

SQL Server

By default, SQL Server uses locks for implementing all the isolation levels stipulated by the SQL standard.

For the *Read Committed* isolation level to take advantage of the MVCC model, the following configuration must be set first:

```
ALTER DATABASE high_performance_java_persistence
SET READ_COMMITTED_SNAPSHOT ON;
```

For a higher-level isolation, SQL Server offers the *Snapshot* isolation mode, which must be activated at the database level:

```
ALTER DATABASE high_performance_java_persistence
SET ALLOW_SNAPSHOT_ISOLATION ON;
```

Because Snapshot is a custom isolation level, it must also be set at the connection level prior to starting a new transaction:

```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT;
GO
BEGIN TRANSACTION;
GO
COMMIT TRANSACTION;
GO
```

After enabling row versioning, the database can track record changes in the *tempdb* database.

When a row is either updated or deleted, the current row entry holds a reference back to the previous version, which is recorded in the version store, in the tempdb database. Rows are not deleted right away but only marked for deletion, the actual removal being done by the *Ghost cleanup task*. Old versions must be kept for as long as a currently running transaction might need them, which is specified by the transaction isolation level.

The Ghost cleanup task runs periodically and reclaims storage from old versions that are no longer necessary.

A long-running transaction would require the database engine to keep some old version for a very long time, and, because version changes are chained in a linked list structure, restoring a very old version might becomes resource intensive.

PostgreSQL

Unlike all other database systems, PostgreSQL stores both the current rows and their previous versions (even the ones for the aborted transactions) in the actual database table. Like Oracle, PostgreSQL embraces the MVCC data access model, and it does not offer a 2PL transaction isolation implementation at all.

Each table row has two additional columns (*xmin* and *xmax*), which are used to control the visibility of various row versions. When a row is inserted, the current transaction identifier is stored in the xmin column.

Both the update and the delete operations end up creating a new row entry with a xmax column storing the current transaction identifier.

The *Vacuum* cleaner process runs regularly and reclaims storage occupied by deleted entries (and successfully committed) or by previous versions that are no longer required by the currently running transactions.

Although PostgreSQL is seen as a pure MVCC model, locking is still required to prevent write-write conflicts or for explicit locking[a]. `SELECT FOR UPDATE` is used to acquire an exclusive row-level lock, while `SELECT FOR SHARE` is for applying a shared lock instead.

[a] http://www.postgresql.org/docs/current/static/explicit-locking.html

MySQL

The InnoDB storage engine offers support for ACID transactions and uses MVCC for controlling access to shared resources. The InnoDB MVCC implementation is very similar to Oracle, and previous versions of database rows are stored in the rollback segment as well.

When a transaction demands a previous row version, MySQL must reconstruct it from rollback segments. Delete operations just mark an entry as being ready for deletion, and the *purge* thread is going to do the actual physical cleanup.

Both the transaction rollback and the previous row version restoring processes (required by a given transaction visibility guarantees) are very much the same thing.

Like other database systems, MySQL also offers explicit locking[a] for when MVCC is no longer satisfactory. A shared lock is acquired using `SELECT LOCK IN SHARE MODE`, while, for exclusive locks, the much more common `SELECT FOR UPDATE` syntax is being used.

[a] https://dev.mysql.com/doc/refman/5.7/en/innodb-locking-reads.html

1.3.2 Phenomena

For reasonable transaction throughput values, it makes sense to imply transaction *serializability*. As the incoming traffic grows, the price for strict data integrity becomes too high, and this is the primary reason for having multiple isolation levels. Relaxing serializability guarantees may generate data integrity anomalies, which are also referred as *phenomena*.

The SQL-92 standard introduced three phenomena that can occur when moving away from a serializable transaction schedule:

- *dirty read*
- *non-repeatable read*
- *phantom read.*

In reality, there are other phenomena that can occur due to transaction interleaving, as the famous paper A *Critique of* ANSI SQL *Isolation Levels*[4] describes:

- *dirty write*
- *read skew*
- *write skew*
- *lost update*.

> Choosing a certain isolation level is a trade-off between increasing concurrency and acknowledging the possible anomalies that might occur.
>
> Scalability is undermined by contention and coherency costs. The lower the isolation level, the less locking (or multi-version transaction abortions), and the more scalable the application gets.
>
> However, a lower isolation level allows more phenomena, and the data integrity responsibility is shifted from the database side to the application logic, which must ensure that it takes all measures to prevent or mitigate any such data anomaly.

Before jumping to isolation levels, it is better to understand what's behind each particular phenomenon and how it can affect data integrity. When choosing a given transaction isolation level, understanding phenomena becomes fundamental to taking the right decision,

[4] http://research.microsoft.com/apps/pubs/default.aspx?id=69541

1.3.2.1 Dirty write

A dirty write happens when two concurrent transactions are allowed to modify the same row at the same time. As previously mentioned, all changes are applied to the actual database object structures, which means that the second transaction simply overwrites the first transaction pending change.

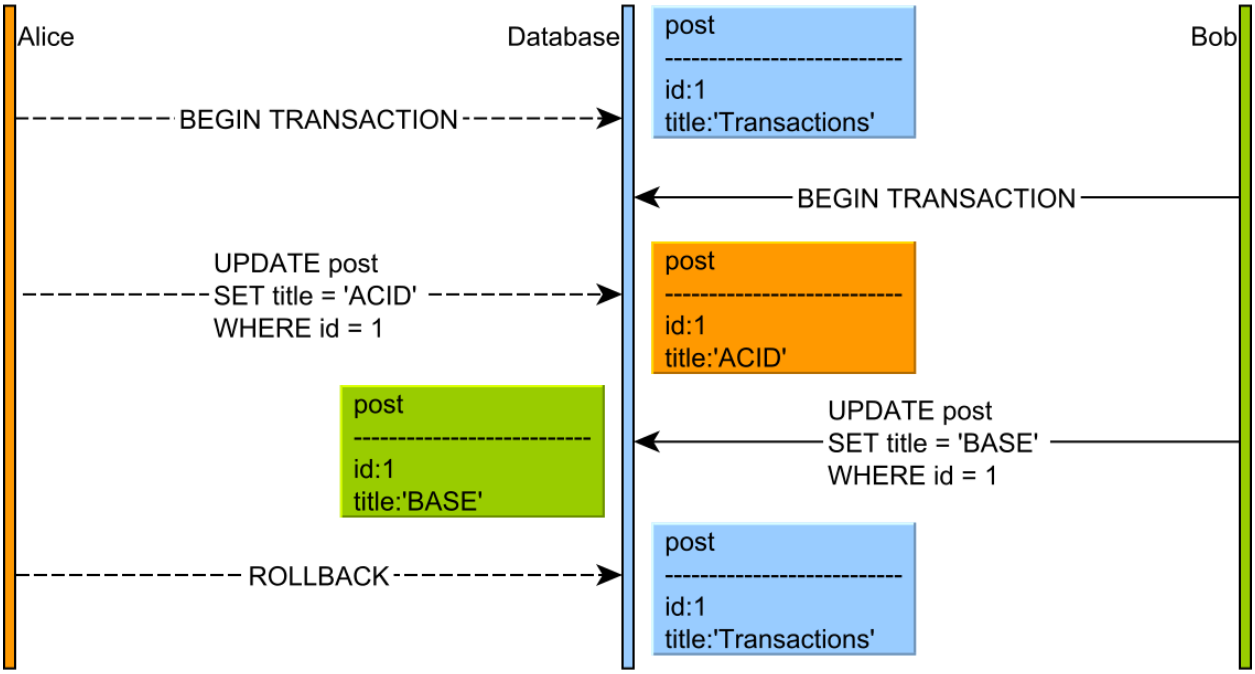

Figure 6.4: Dirty write

If the two transactions commit, one transaction will silently overwrite the other transaction, causing a *lost update*. Another problem arises when the first transaction wants to roll back. The database engine would have to choose one of the following action paths:

- It can restore the row to its previous version (as it was before the first transaction changed it), but then it overwrites the second transaction uncommitted change.
- It can acknowledge the existence of a newer version (issued by the second transaction), but then, if the second transaction has to roll back, its previous version will become the uncommitted change of the first transaction.

If the database engine did not prevent dirty writes, guaranteeing rollbacks would not be possible. Because atomicity cannot be implemented in the absence of reliable rollbacks, all database systems must, therefore, prevent dirty writes.

Although the SQL standard does not mention this phenomenon, even the lowest isolation level (*Read Uncommitted*) can prevent it.

1.3.2.2 Dirty read

As previously mentioned, all database changes are applied to the actual data structures (memory buffers, data blocks, indexes). A dirty read happens when a transaction is allowed to read the uncommitted changes of some other concurrent transaction. Taking a business decision on a value that has not been committed is risky because uncommitted changes might get rolled back.

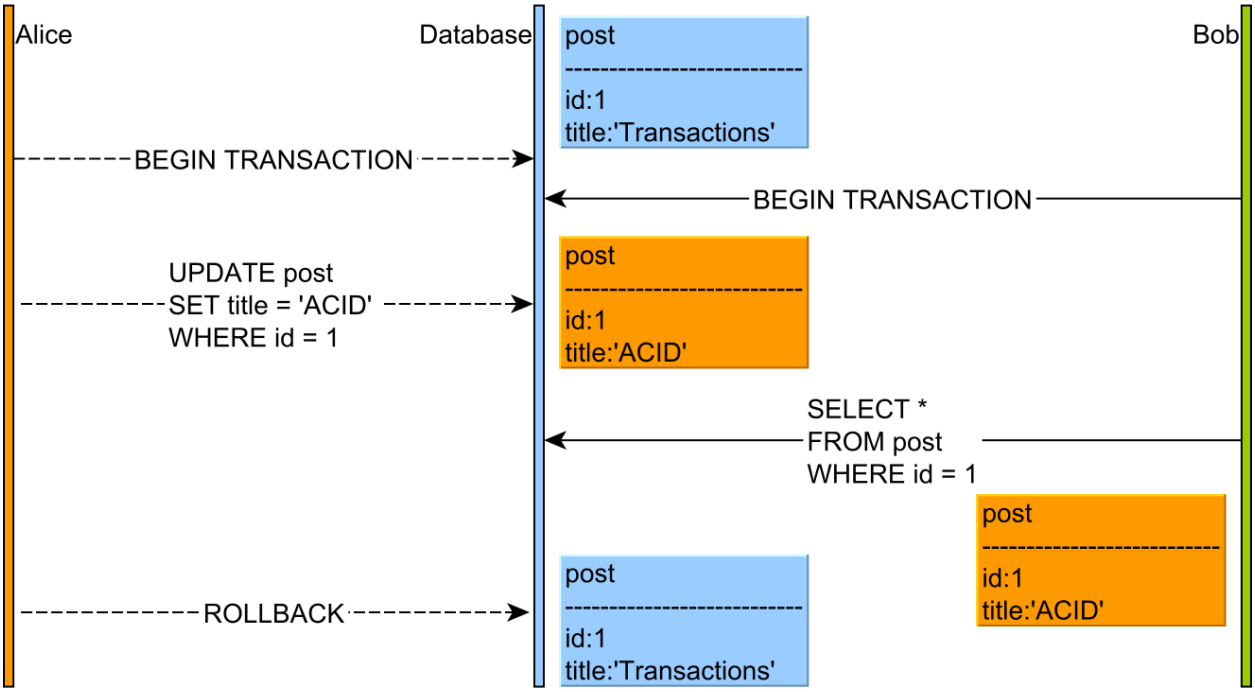

Figure 6.5: Dirty read

This anomaly is only permitted by the Read Uncommitted isolation level, and, because of the serious impact on data integrity, most database systems offer a higher default isolation level.

To prevent dirty reads, the database engine must hide uncommitted changes from all other concurrent transactions. Each transaction is allowed to see its own changes because otherwise the read-your-own-writes consistency guarantee is compromised.

If the underlying database uses 2PL (Two-Phase Locking), the uncommitted rows are protected by write locks which prevent other concurrent transactions from reading these records until they are committed.

When the underlying database uses MVCC (Multi-Version Concurrency Control), the database engine can use the undo log which already captures the previous version of every uncommitted record, to restore the previous value in other concurrent transaction queries. Most database systems optimize the before image restoring process, therefore lowering its overhead on the overall application performance.

Read Uncommitted is rarely needed (non-strict reporting queries where dirty reads are acceptable), so Read Committed is usually the lowest practical isolation level.

1.3.2.3 Non-repeatable read

If one transaction reads a database row without applying a shared lock on the newly fetched record, then a concurrent transaction might change this row before the first transaction has ended.

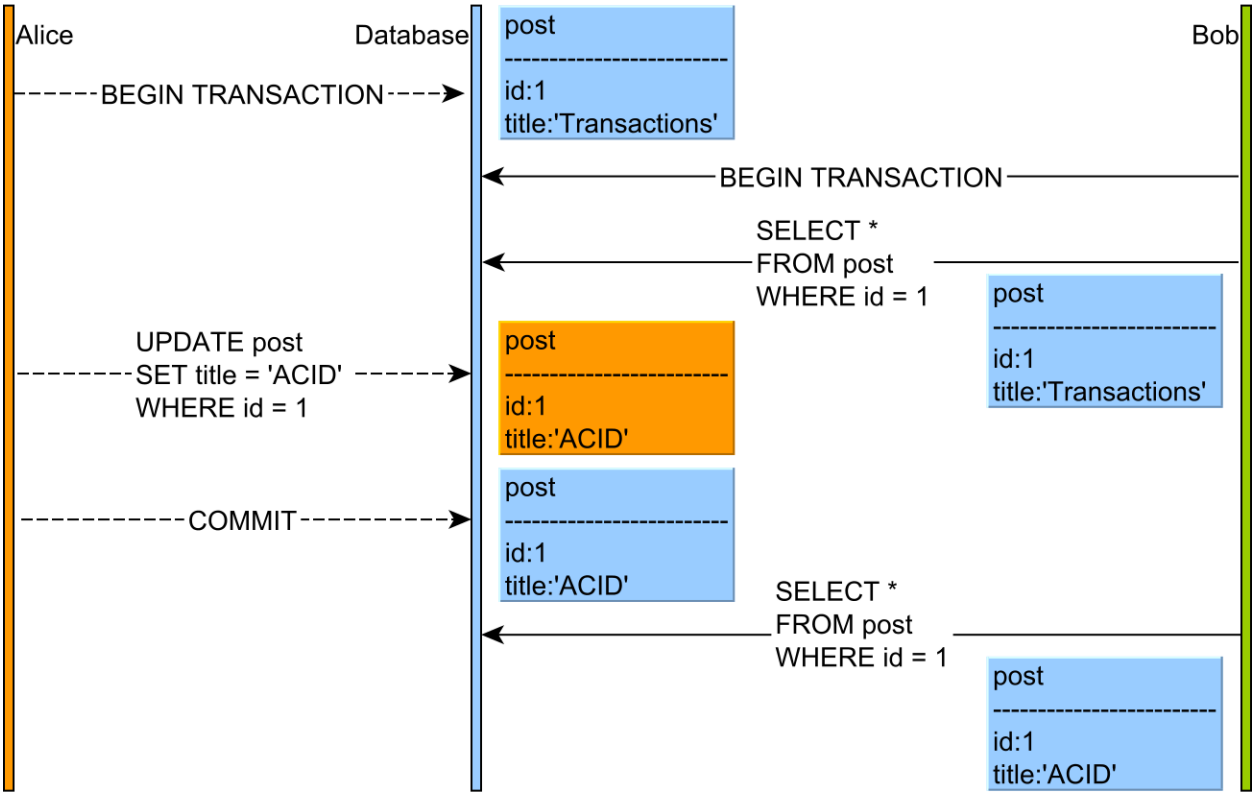

Figure 6.6: Non-repeatable read

This phenomenon is problematic when the current transaction makes a business decision based on the first value of the given database row (a client might order a product based on a stock quantity value that is no longer a positive integer).

Most database systems have moved to a Multi-Version Concurrency Control model, and shared locks are no longer mandatory for preventing non-repeatable reads. By verifying the current row version, a transaction can be aborted if a previously fetched record has changed in the meanwhile.

Repeatable Read and Serializable prevent this anomaly by default. With Read Committed, it is possible to avoid non-repeatable (fuzzy) reads if the shared locks are acquired explicitly (e.g. `SELECT FOR SHARE`).

Some ORM frameworks (e.g. JPA/Hibernate) offer application-level repeatable reads. The first snapshot of any retrieved entity is cached in the currently running *Persistence Context*. Any successive query returning the same database row is going to use the very same object that was previously cached. This way, the fuzzy reads may be prevented even in Read Committed isolation level.

1.3.2.4 Phantom read

If a transaction makes a business decision based on a set of rows satisfying a given predicate, without range locks, a concurrent transaction might insert a record matching that particular predicate.

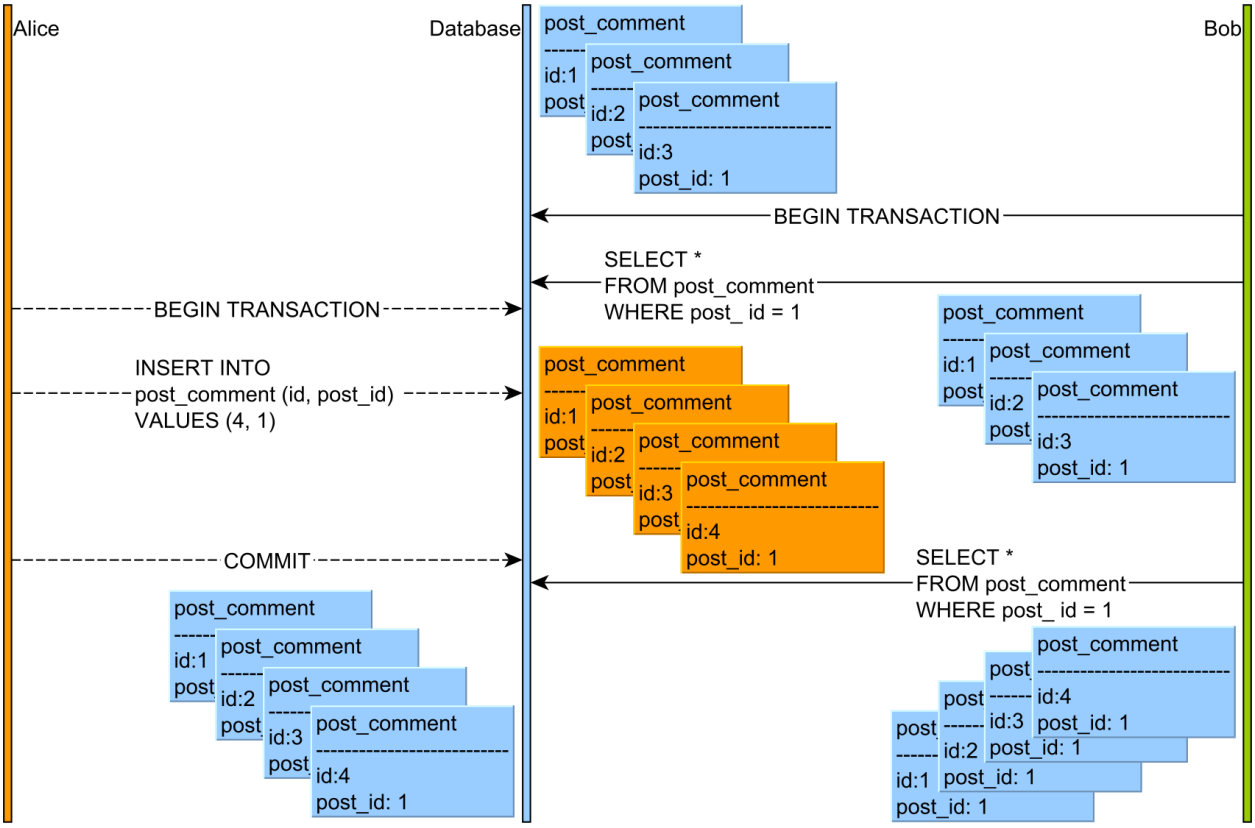

Figure 6.7: Phantom read

The SQL standard says that Phantom Read occurs if two consecutive query executions render different results because a concurrent transaction has modified the range of records in between the two calls. Although providing consistent reads is a mandatory requirement for serializability, that is not sufficient. For instance, one buyer might purchase a product without being aware of a better offer that was added right after the user has finished fetching the offer list.

The 2PL-based Serializable isolation prevents Phantom Reads through the use of predicate locking. On the other hand, MVCC database engines address the Phantom Read anomaly by returning consistent snapshots. However, a concurrent transaction can still modify the range of records that was read previously. Even if the MVCC database engine introspects the transaction schedule, the outcome is not always the same as a 2PL-based implementation. One such example is when the second transaction issues an insert without reading the same range of records as the first transaction. In this particular use case, some MVCC database engines will not end up rolling back the first transaction.

1.3.2.5 Read skew

Read skew is a lesser known anomaly that involves a constraint on more than one database tables. In the following example, the application requires the *post* and the *post_details* be updated in sync. Whenever a *post* record changes, its associated *post_details* must register the user who made the current modification.

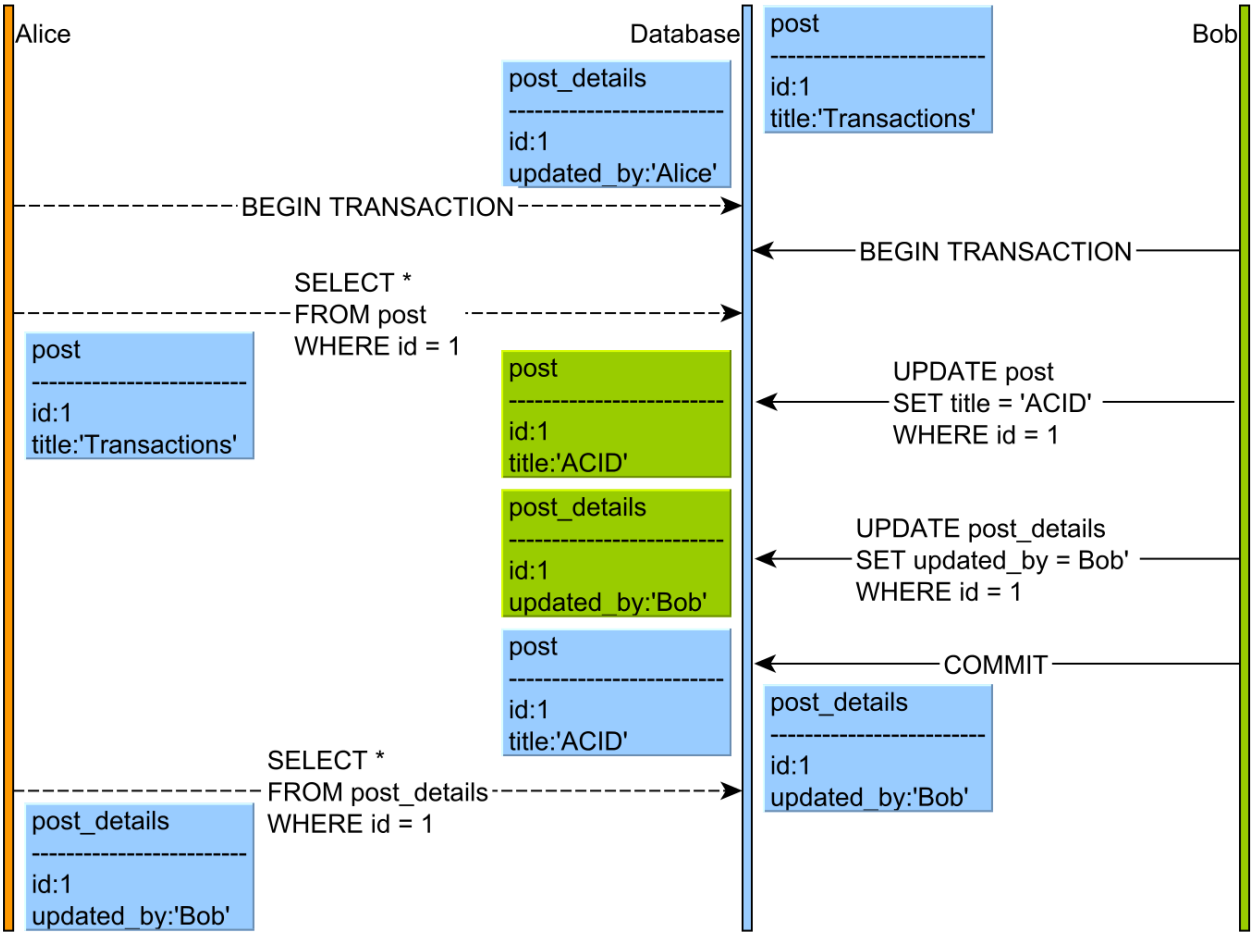

Figure 6.8: Read skew

In between selecting the *post* and the *post_details* rows, a second transaction sneaks in and manages to update both records. The first transaction sees an older version of the *post* row and the latest version of the associated *post_details*. Because of this read skew, the first transaction assumes that this particular *post* was updated by *Bob*, although, in fact, it is an older version updated by *Alice*.

Like with non-repeatable reads, there are two ways to avoid this phenomenon:

- The first transaction can acquire shared locks on every read, therefore preventing the second transaction from updating these records.
- The first transaction can be aborted upon validating the commit constraints (when using an MVCC implementation of the Repeatable Read or Serializable isolation levels).

1.3.2.6 Write skew

Like read skew, this phenomenon involves disjoint writes over two different tables that are constrained to be updated as a unit. Whenever the *post* row changes, the client must update the *post_details* with the user making the change.

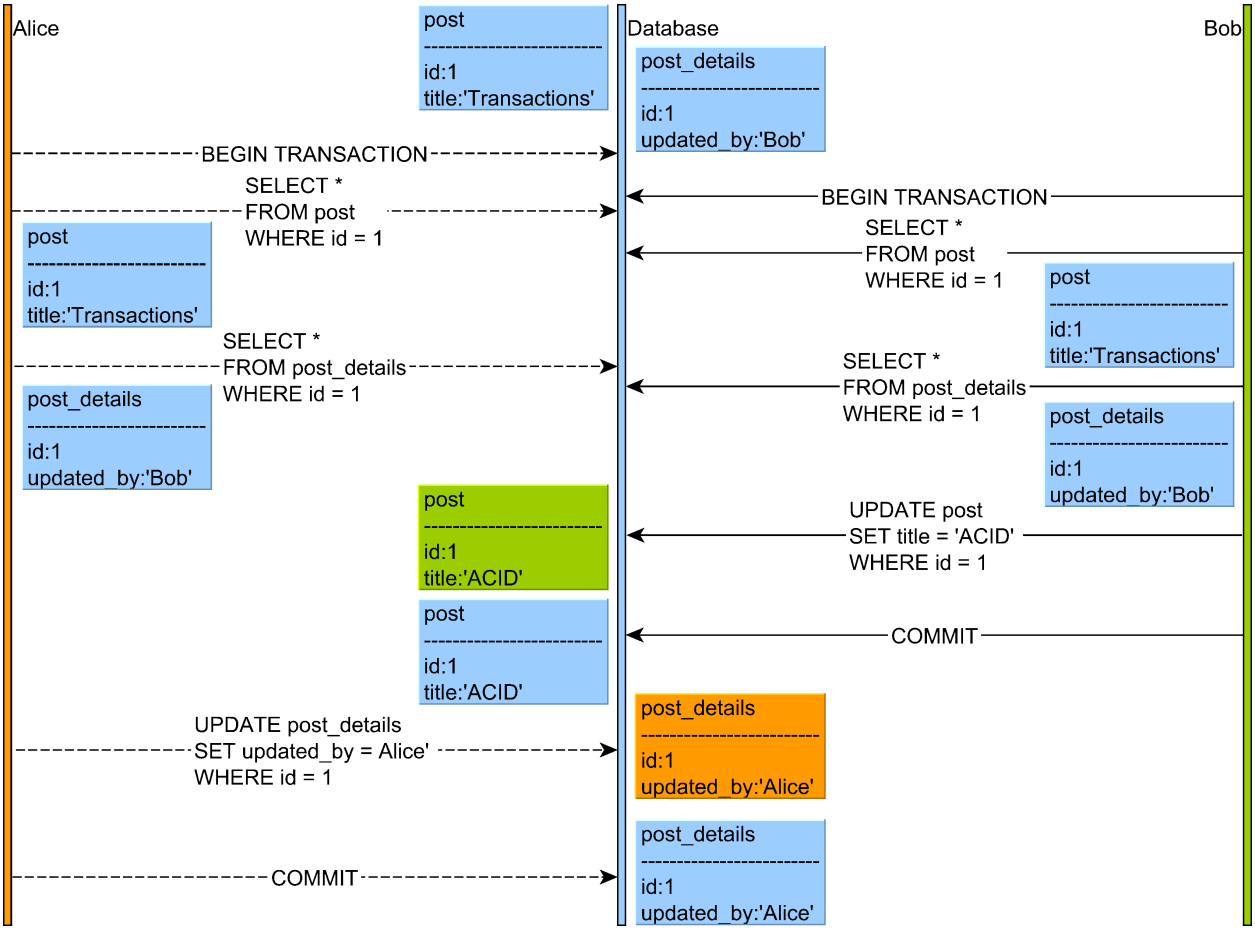

Figure 6.9: Write skew

Both *Alice* and *Bob* selects the *post* and its associated *post_details* record. If write skew is allowed, *Alice* and *Bob* can update these two records separately, therefore breaking the constraint.

Like with non-repeatable reads, there are two ways to avoid this phenomenon:

- The first transaction can acquire shared locks on both entries, therefore preventing the second transaction from updating these records.
- The database engine can detect that another transaction has changed these records, and so it can force the first transaction to roll back (under an MVCC implementation of Repeatable Read or Serializable).

1.3.2.7 Lost update

This phenomenon happens when a transaction reads a row while another transaction modifies it prior to the first transaction to finish. In the following example, *Bob*'s update is silently overwritten by *Alice*, who is not aware of the record update.

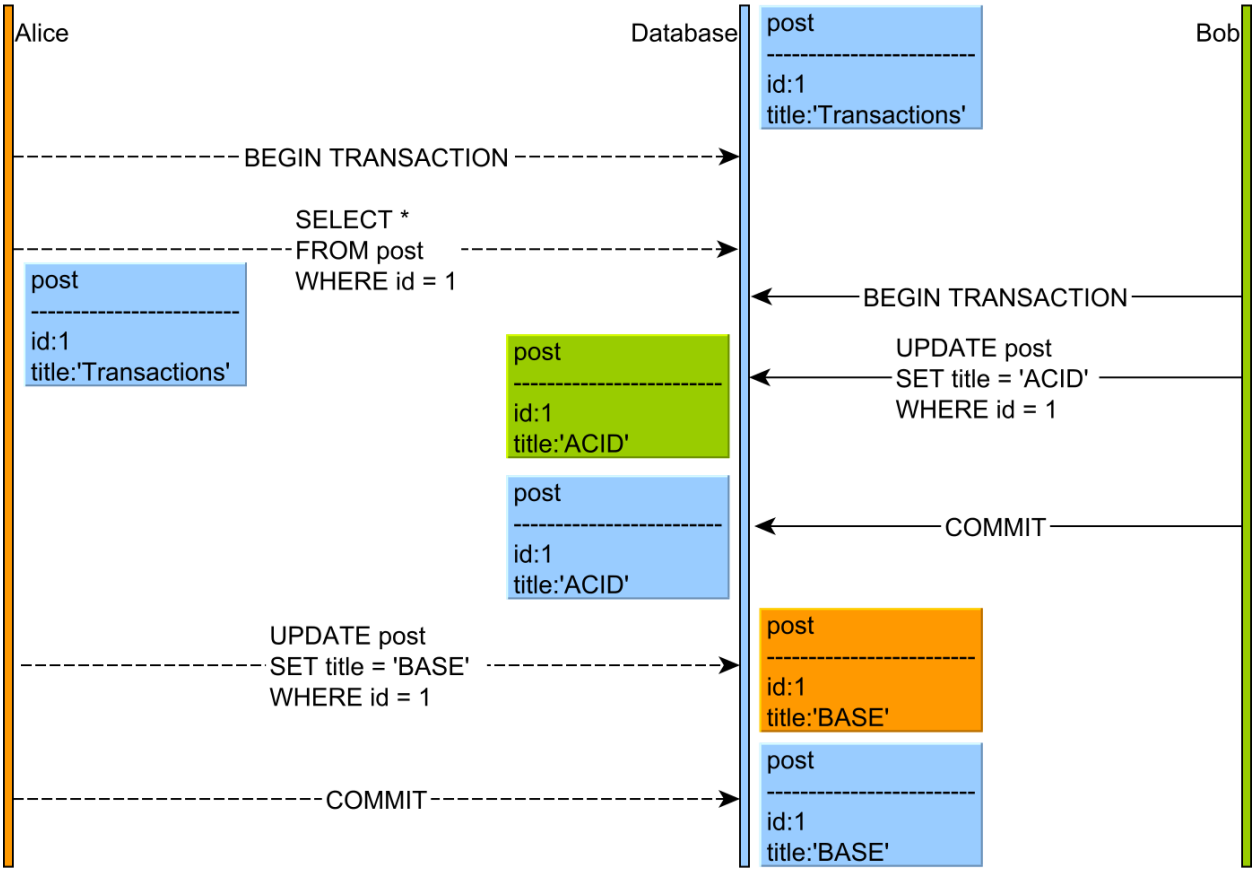

Figure 6.10: Lost update

This anomaly can have serious consequences on data integrity (a buyer might purchase a product without knowing the price has just changed), especially because it affects Read Committed, the default isolation level in many database systems.

Traditionally, Repeatable Read protected against lost updates since the shared locks could prevent a concurrent transaction from modifying an already fetched record. With MVCC, the second transaction is allowed to make the change, while the first transaction is aborted when the database engine detects the row version mismatch (during the first transaction commit).

Most ORM tools, such as Hibernate, offer application-level optimistic locking, which automatically integrates the row version whenever a record modification is issued. On a row version mismatch, the update count is going to be zero, so the application can roll back the current transaction, as the current data snapshot is stale.

1.3.3 Isolation levels

As previously stated, Serializable is the only isolation level to provide a truly ACID transaction interleaving. However, serializability comes at a price as locking introduces contention, which, in turn, limits concurrency and scalability. Even in multi-version concurrency models, serializability may require aborting too many transactions that are affected by phenomena.

For this purpose, the SQL-92 version introduced multiple isolation levels, and the database client has the option of balancing concurrency against data correctness. Each isolation level is defined in terms of the minimum number of phenomena that it must prevent, and so the SQL standard introduces the following transaction isolation levels:

Table 6.1: Standard isolation levels

Isolation Level	Dirty read	Non-repeatable read	Phantom read
Read Uncommitted	Yes	Yes	Yes
Read Committed	No	Yes	Yes
Repeatable Read	No	No	Yes
Serializable	No	No	No

Without an explicit setting, the JDBC driver uses the default isolation level, which can be introspected using the `getDefaultTransactionIsolation()`[5] method of the `DatabaseMetaData` object:

```
int level = connection.getMetaData().getDefaultTransactionIsolation();
```

The default isolation level can be changed using the `setTransactionIsolation(int level)`[6] `Connection` method.

```
connection.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
```

> Even if ACID properties imply a serializable schedule, most relational database systems use a lower default isolation level instead:
>
> - Read Committed (Oracle, SQL Server, PostgreSQL)
> - Repeatable Read (MySQL).

The following sections go through each particular transaction isolation level and demonstrate the actual list of phenomena that are prevented by a given database system.

[5] http://docs.oracle.com/javase/8/docs/api/java/sql/DatabaseMetaData.html#getDefaultTransactionIsolation--
[6] http://docs.oracle.com/javase/8/docs/api/java/sql/Connection.html#setTransactionIsolation-int-

1.3.3.1 Read Uncommitted

Table 6.2: Read Uncommitted phenomena occurrence

Phenomena	SQL Server	PostgreSQL	MySQL
Dirty Write	No	No	No
Dirty Read	Yes	No	Yes
Non-Repeatable Read	Yes	Yes	Yes
Phantom Read	Yes	Yes	Yes
Read Skew	Yes	Yes	Yes
Write Skew	Yes	Yes	Yes
Lost Update	Yes	Yes	Yes

Oracle

Dirty reads are not allowed, and so the lowest isolation level is Read Committed.

The JDBC driver will even throw an exception if the client tries to set the Read Uncommitted isolation on the current `Connection`.

SQL Server

Read Uncommitted only protects against dirty writes, all other phenomena being allowed. When using Read Uncommitted, there is no exclusive lock associated with a given SQL modification, so uncommitted changes are available to other concurrent transactions even before they get committed. If the risk of dirty reads can be assumed, avoiding exclusive locks may speed up reporting queries, especially when scanning large amounts of data.

For lock-based concurrency control mechanisms, Read Uncommitted is worth considering if the risk of dirty reads is a much smaller issue than locking a large portion of a database table. Because MVCC avoids reader-writer and writer-reader locking, it might not exhibit a considerable performance enhancement from permitting dirty reads.

PostgreSQL

Like Oracle, PostgreSQL does not allow dirty reads, the lowest isolation level being Read Committed.

When choosing Read Uncommitted, the JDBC driver silently falls back to Read Committed.

MySQL

Although it uses MVCC, InnoDB implements Read Uncommitted so that dirty reads are permitted. As an optimization, each query is spared from rebuilding the previously committed versions (using the rollback segments) of the currently scanned records (in case they have been recently modified).

1.3.3.2 Read Committed

Read Committed is one of the most common isolation levels, and it behaves consistently across multiple relational database systems or various concurrency control models.

Many database systems choose Read Committed as the default isolation level because it delivers the best performance while preventing fatal anomalies such as dirty writes and dirty reads. However, performance has its price as Read Committed permits many anomalies that might lead to data corruption.

Table 6.3: Read Committed phenomena occurrence

Phenomena	Oracle	SQL Server	SQL Server MVCC	PostgreSQL	MySQL
Dirty Write	No	No	No	No	No
Dirty Read	No	No	No	No	No
Non-Repeatable Read	Yes	Yes	Yes	Yes	Yes
Phantom Read	Yes	Yes	Yes	Yes	Yes
Read Skew	Yes	Yes	Yes	Yes	Yes
Write Skew	Yes	Yes	Yes	Yes	Yes
Lost Update	Yes	Yes	Yes	Yes	Yes

Oracle

Every statement has a start timestamp, which is used to create a database snapshot relative to this particular point-in-time. This way, writers can still update the currently selected records, and the database can simply reconstruct the previous versions that were available when the query started. Subsequent query executions can return different row versions, so non-repeatable reads are permitted.

When two transactions attempt to update the same record, the first one locks the record to prevent dirty writes. The second transaction must wait until the first transaction releases the lock (either commit or rollback), and the statement filtering criteria are reevaluated against latest data.

PostgreSQL

Like Oracle, every query sees a database snapshot as of the beginning of the currently running query. Because shared locks are not used to protect previously read records from being modified, Read Committed allows a large spectrum of data anomalies.

Exclusive locks prevent write-write conflicts, so when two transactions update the same record, the second one waits for the first transaction to release its locks. When the second transaction resumes its execution, if the filtering criteria are still relevant, it might overwrite the first transaction modifications, therefore causing lost updates.

MySQL

Query-time snapshots are used to isolate statements from other concurrent transactions. When explicitly acquiring shared or exclusive locks or when issuing update or delete statements (which acquire exclusive locks to prevent dirty writes), if the selected rows are filtered by unique search criteria (e.g. primary key), the locks can be applied to the associated index entries.

Prior to 5.7[a], if the modifying statements used a range filter and the search criteria took advantage of a unique index scan, then the database could use a gap or a next-key lock (therefore protecting against phantom reads as well). Statement-based replication is not available for Read Committed, so the application must use the row-based binary logging instead.

[a] https://dev.mysql.com/doc/refman/5.7/en/set-transaction.html#idm140311316367072

SQL Server

By default, SQL statements use shared locks to prevent other transactions from modifying the currently fetched records. The locks are released by the time the query finishes executing. When activating Read Committed Snapshot Isolation, the database does not use shared locks anymore, and each query selects the row version as it was when the query started.

1.3.3.3 Repeatable Read

One of the least compliant isolation levels, Repeatable Read implementation details leak into its phenomena prevention spectrum:

Table 6.4: Repeatable Read phenomena occurrence

Phenomena	SQL Server	PostgreSQL	MySQL
Dirty Write	No	No	No
Dirty Read	No	No	No
Non-Repeatable Read	No	No	No
Phantom Read	Yes	No	No
Read Skew	No	No	No
Write Skew	No	Yes	Yes
Lost Update	No	No	Yes

Oracle

The Repeatable Read isolation is not supported at all, and the JDBC driver will throw an exception if the client tries to set it explicitly.

SQL Server

For every row the client reads, the current transaction acquires a shared lock that prevents any other transaction from concurrently modifying it. The shared locks are released when the transaction either commits or rolls back.

PostgreSQL

The Repeatable Read is implemented using Snapshot Isolation[a], so not only fuzzy reads are prevented, but even phantom reads are prohibited as well. Instead of using locking, the PostgreSQL MVCC implementation allows conflicts to occur, but it aborts any transaction whose guarantees do not hold anymore.

[a] https://en.wikipedia.org/wiki/Snapshot_isolation

MySQL

Every transaction can only see rows as if they were when the current transaction started. This prevents non-repeatable reads, but it still allows lost updates and write skews.

1.3.3.4 Serializable

Serializable is supposed to provide a transaction schedule, whose outcome, even in spite of statement interleaving, is equivalent to a serial execution.

Even if the concurrency control mechanism is lock-based or it manages multiple record versions, it must prevent all phenomena to ensure serializable transactions. Preventing all phenomena mentioned by the SQL standard (dirty reads, non-repeatable reads and phantom reads) is not enough, and Serializable must protect against lost update, read skew and write skew as well.

In practice, the concurrency control implementation details leak, and not all relational database systems provide a truly Serializable isolation level (some data integrity anomalies might still occur).

Table 6.5: Serializable phenomena occurrence

Phenomena	Oracle	SQL Server	SQL Server MVCC	PostgreSQL	MySQL
Dirty Write	No	No	No	No	No
Dirty Read	No	No	No	No	No
Non-Repeatable Read	No	No	No	No	No
Phantom Read	No	No	No	No	No
Read Skew	No	No	No	No	No
Write Skew	Yes	No	Yes	No	No
Lost Update	No	No	No	No	No

Oracle

The Serializable isolation level is, in fact, an MVCC implementation of the Snapshot Isolation concurrency control mechanism. Like the Repeatable Read isolation in PostgreSQL, Oracle cannot prevent write skews, meaning it cannot provide a truly serializable transaction.

SQL Server

The Serializable isolation level is based on 2PL, and all phenomena are therefore prevented. The MVCC-based Snapshot Isolation is close to Oracle Serializable and PostgreSQL Repeatable Read, and so it allows write skews.

PostgreSQL

To overcome the Snapshot Isolation limitations, PostgreSQL has developed the Serializable Snapshot Isolation (SSI), which provides true serializable transactions. Because SSI is still an MVCC implementation, PostgreSQL monitors the transaction schedule and detects *possible* serializability anomalies.

The current implementation may detect *false positives*[a], and some transactions might get aborted even if they did not really break transaction serializability. Only the Precisely Serializable Snapshot Isolation (PSSI) model can eliminate all false positives, but the performance penalty being too high, the database implementers stuck to SSI instead.

[a] http://drkp.net/papers/ssi-vldb12.pdf

MySQL

The Serializable isolation builds on top of Repeatable Read with the difference that every record that gets selected is protected with a shared lock as well. The lock-based approach allows MySQL to prevent the write skew phenomena, which is prevalent among many Snapshot Isolation implementations.

1.4 Durability

When purchasing an airline ticket, the money is withdrawn from the bank account, and a seat is reserved for the given buyer. Assuming that, right after the ticket is purchased, the airline reservation system crashes, all the previously processed transactions must hold true even after the system restarts. If the system does not enforce this requirement, the registered ticket might vanish, and the buyer is possibly left with a debited account and no ticket at all.

> Durability ensures that all committed transaction changes become permanent.

Durability allows system recoverability, and, to some extent, it is similar to the rolling back mechanism.

What about undo logs?

To support transaction rollbacks and to rebuild previous versions in MVCC systems, the database system already records the current modifications (including uncommitted changes) in the undo log. However, recoverability needs committed changes only, and, because the obsolete undo segments might be frequently recycled, the undo log alone is not suitable for recoverability.

When a transaction is committed, the database persists all current changes in an append-only, sequential data structure commonly known as the *redo log*.

Oracle

The *redo log*[a] consists of multiple redo records, each one containing *change vectors*, which capture the actual data block changes. For performance reasons, the redo records are stored in a buffer, and the Log Writer flushes the in-memory records to the current active redo log file. At any given time, Oracle has, at least, two redo files, but only one of them is active and available for collecting the log buffer entries. When a transaction is committed, the database flushes the buffer and changes become persisted.

If the buffer fills, Oracle will flush it along with any uncommitted changes, which can be removed if their associated transaction is rolled back.

[a] https://docs.oracle.com/database/121/ADMIN/onlineredo.htm#ADMIN11302

SQL Server

Unlike Oracle, SQL Server combines both the undo and the redo log into a single data structure called *transaction log*. By default, when a transaction is committed, all the associated transaction log entries are flushed to the disk before returning the control back to the client.

SQL Server 2014 added support for configurable durability[a]. The log entry flushing can be delayed, which can provide better I/O utilization and lower transaction response times. If the system crashes, all the unflushed log entries will be wiped out from memory. Asynchronous flushing is, therefore, appropriate only when data loss is tolerated.

[a] https://msdn.microsoft.com/en-us/library/dn449490.aspx

PostgreSQL

Statement changes are captured in the *Write-Ahead Log (WAL)*[a]. The log entries are buffered in memory and flushed on every transaction commit.

Because their state can be restored from the WAL during recovery, the cached data pages and index entries need not be flushed for every transaction (therefore optimizing I/O utilization). Ever since 9.1[b], PostgreSQL supports configurable durability, so the WAL can also be flushed asynchronously.

[a] http://www.postgresql.org/docs/current/static/wal-intro.html
[b] http://www.postgresql.org/docs/current/static/non-durability.html

MySQL

All the *redo log* entries associated with a single transaction are stored in the *mini transaction buffer* and flushed at once into the *global redo buffer*. The global buffer is flushed to disk during commit. By default, there are two log files which are used alternatively.

Flushing is done synchronously by default, but it can be switched to an asynchronous mode via the innodb_flush_log_at_trx_commit[a] parameter.

[a] http://dev.mysql.com/doc/refman/5.7/en/innodb-parameters.html#sysvar_innodb_flush_log_at_trx_commit

Since durability is very important for business operations, it is better to stick to the synchronous flushing mechanism.

Delaying durability guarantees becomes a valid option only when data loss is tolerated by business requirements and the redo log flushing is a real performance bottleneck.

1.5 Read-only transactions

The JDBC `Connection` defines the `setReadOnly(boolean readOnly)`[7] method which can be used to *hint* the driver to apply some database optimizations for the upcoming read-only transactions. This method should not be called in the middle of a transaction because the database system cannot turn a read-write transaction into a read-only one (a transaction must start as read-only from the very beginning).

Oracle

According to the JDBC driver documentation[a], the database server does not support read-only transaction optimizations. Even when the read-only `Connection` status is set to `true`, modifying statements are still permitted, and the only way to restrict such statements is to execute the following SQL command:

```
connection.setAutoCommit(false);
try(CallableStatement statement = connection.prepareCall(
   "BEGIN SET TRANSACTION READ ONLY; END;")) {
    statement.execute();
}
```

The `SET TRANSACTION READ ONLY` command must run after disabling the auto-commit status, as otherwise it is only applied for this particular statement only.

[a] https://docs.oracle.com/database/121/JJDBC/apxtips.htm#JJDBC28956

[7] http://docs.oracle.com/javase/8/docs/api/java/sql/Connection.html#setReadOnly%28boolean%29

SQL Server

Like Oracle, the read-only `Connection` does not propagate to the database engine, and the only way to disable SQL modifications is to use a separate account, restricted to viewing data only.

Setting the `ApplicationIntent=ReadOnly`[a] connection property does not prevent the JDBC driver from executing modifying statements on a read-only `Connection`. This property has the purpose of routing read-write and read-only connections to replica nodes instead.

[a] https://msdn.microsoft.com/en-us/library/gg471494.aspx

PostgreSQL

An exception is thrown when executing a modifying statement on a `Connection` whose read-only status was set to `true`.

The database engine optimizes read-only transactions, so the false-positives anomaly rate is reduced for the Serializable isolation level, and it allows *deferrable* serializable snapshots[a]. A deferrable snapshot is activated when executing `SET TRANSACTION SERIALIZABLE READ ONLY DEFERRABLE`. The current transaction must wait for a safe snapshot to become available, which can be executed without the risk of being aborted by a non-serializable anomaly. If the default read-write Serializable transactions are problematic when accessing large volumes of data, the deferrable snapshots might be a better alternative for long-running transactions.

[a] http://arxiv.org/pdf/1208.4179.pdf

MySQL

If a modifying statement is executed when the `Connection` is set to read-only, the JDBC driver will throw an exception.

InnoDB can optimize read-only transactions[a] because it can skip the transaction ID generation as it is not required for read-only transactions.

[a] https://dev.mysql.com/doc/refman/5.7/en/innodb-performance-ro-txn.html

1.5.1 Read-only transaction routing

Setting up a database replication environment is useful for both high-availability (a Slave can replace a crashing Master) and traffic splitting. In a Master-Slave replication topology, the Master node accepts both read-write and read-only transactions, while Slave nodes only take read-only traffic.

Oracle

The Oracle ADG (Active Data Guard) allows an enterprise application to distribute read-write traffic to the Primary node and read-only transactions to a Standby database. WebLogic Server GridLink Data Source[a] provides failover and load balancing capabilities over Oracle ADG.

[a] http://www.oracle.com/technetwork/middleware/weblogic/learnmore/1534212

SQL Server

The database Availability Group must be configured to use read-only routing, in which case the redirection is based on the `ApplicationIntent` connection property. This means that the application requires separate `DataSource(s)` for read-write and read-only connections, and transaction routing must initiate in the application service layer.

PostgreSQL

The JDBC driver defines two connection properties[a] for load balancing purposes: `loadBalanceHosts` (which is disabled by default) and `targetServerType` (`master` or `preferSlave`). To enable transaction routing, the application must do the routing itself using separate `DataSource(s)`.

[a] https://jdbc.postgresql.org/documentation/head/connect.html

MySQL

The `com.mysql.jdbc.ReplicationDriver`[a] supports transaction routing on a Master-Slave topology, the decision being made on the `Connection` read-only status basis.

[a] https://dev.mysql.com/doc/connector-j/en/connector-j-master-slave-replication-connection.html

Even if the JDBC driver does not support Master-Slave routing, the application can do it using multiple `DataSource` instances. This design cannot rely on the read-only status of the underlying `Connection` since the routing must take place before a database connection is fetched.

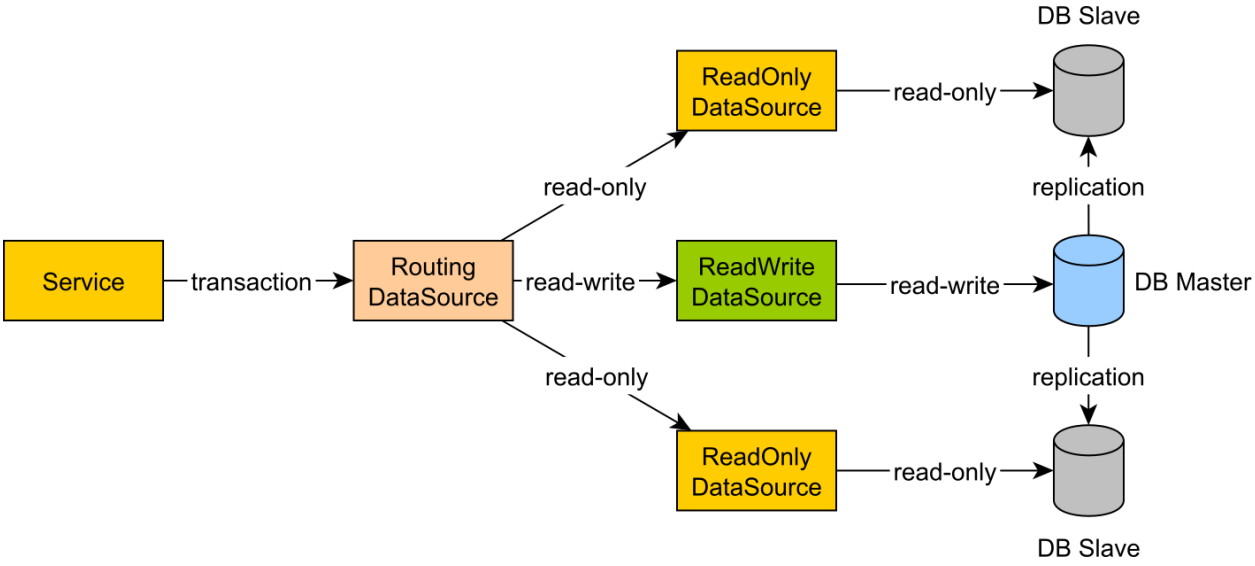

Figure 6.11: Transaction routing

If the transaction manager supports declarative read-only transactions, the routing decision can be taken based on the current transaction read-only preference. Otherwise, the routing must be done manually in each service layer component, and so a read-only transaction uses a read-only `DataSource` or a read-only JPA Persistence Context.

1.6 Transaction boundaries

Every database statement executes in the context of a database transaction, even if the client does not explicitly set transaction boundaries. While there might be single statement transactions (usually a read-only query), when the unit of work consists of multiple SQL statements, the database should wrap them all in a single unit of work.

By default, every `Connection` starts in *auto-commit* mode, each statement being executed in a separate transaction. Unfortunately, it does not work for multi-statement transactions as it moves atomicity boundaries from the logical unit of work to each individual statement.

Auto-commit should be avoided as much as possible, and, even for single statement transactions, it is good practice to mark the transaction boundaries explicitly.

In the following example, a sum of money is transferred between two bank accounts. The balance must always be consistent, so if an account gets debited, the other one must always be credited with the same amount of money.

```
try(Connection connection = dataSource.getConnection();
    PreparedStatement transferStatement = connection.prepareStatement(
        "UPDATE account SET balance = ? WHERE id = ?"
    )) {
    transferStatement.setLong(1, Math.negateExact(cents));
    transferStatement.setLong(2, fromAccountId);
    transferStatement.executeUpdate();

    transferStatement.setLong(1, cents);
    transferStatement.setLong(2, toAccountId);
    transferStatement.executeUpdate();
}
```

Because of the auto-commit mode, if the second statement failed, only those particular changes could be rolled back, the first statement being already committed cannot be reverted anymore.

The default auto-commit mode must be disabled and the transaction has to be managed manually. The transaction is committed if every statement runs successfully and a rollback is triggered on a failure basis. With this in mind, the previous example should be rewritten as follows:

```
try(Connection connection = dataSource.getConnection()) {
    connection.setAutoCommit(false);
    try(PreparedStatement transferStatement = connection.prepareStatement(
        "UPDATE account SET balance = ? WHERE id = ?"
    )) {
        transferStatement.setLong(1, Math.negateExact(cents));
        transferStatement.setLong(2, fromAccountId);
        transferStatement.executeUpdate();

        transferStatement.setLong(1, cents);
        transferStatement.setLong(2, toAccountId);
        transferStatement.executeUpdate();

        connection.commit();
    } catch (SQLException e) {
        connection.rollback();
        throw e;
    }
}
```

The astute reader might notice that the previous example breaks the *Single responsibility principle* since the Data Access Object (DAO) method mixes both transaction management and data access logic. Transaction management is a cross-cutting concern, making it a good candidate for being moved to a separate common library. This way, the transaction management logic resides in one place, and a lot of duplicated code can be removed from the DAO methods. One way to extract the transaction management logic is to use the *Template method pattern*:

```
public void transact(Consumer<Connection> callback) {
    Connection connection = null;
    try {
        connection = dataSource.getConnection();
        callback.accept(connection);
        connection.commit();
    } catch (Exception e) {
        if (connection != null) {
            try {
                connection.rollback();
            } catch (SQLException ex) {
                throw new DataAccessException(e);
            }
        }
        throw (e instanceof DataAccessException ?
            (DataAccessException) e : new DataAccessException(e));
    } finally {
        if(connection != null) {
            try {
                connection.close();
            } catch (SQLException e) {
                throw new DataAccessException(e);
            }
        }
    }
}
```

Transactions should never be abandoned on failure, and it is mandatory to initiate a transaction rollback (to allow the database to revert any uncommitted changes and release any lock as soon as possible).

With this utility in hand, the previous example can be simplified to:

```
transact((Connection connection) -> {
    try(PreparedStatement transferStatement = connection.prepareStatement(
        "UPDATE account SET balance = ? WHERE id = ?"
    )) {
        transferStatement.setLong(1, Math.negateExact(cents));
        transferStatement.setLong(2, fromAccountId);
        transferStatement.executeUpdate();

        transferStatement.setLong(1, cents);
        transferStatement.setLong(2, toAccountId);
        transferStatement.executeUpdate();
    } catch (SQLException e) {
        throw new DataAccessException(e);
    }
});
```

Although better than the first code snippet, separating data access logic and transaction management is not sufficient.

The transaction boundaries are still rigid, and, to include multiple data access method in a single database transaction, the `Connection` object has to be carried out as a parameter to every single DAO method.

Declarative transactions can better address this issue by breaking the strong coupling between the data access logic and the transaction management code. Transaction boundaries are marked with metadata (e.g. annotations) and a separate transaction manager abstraction is in charge of coordinating transaction logic.

Java EE and JTA

Declarative transactions become a necessity for distributed transactions. When Java EE (Enterprise Edition) first emerged, application servers hosted both web applications and middleware integration services, which meant that the Java EE container needed to coordinate multiple `DataSource(s)` or even JMS (Java Messaging) queues.

Following the X/Open XA architecture, JTA (Java Transaction API) powers the Java EE distributed transactions requirements.

1.6.1 Distributed transactions

The difference between local and global transactions is that the former uses a single resource manager, while the latter operates on multiple heterogeneous resource managers. The ACID guarantees are still enforced on each individual resource, but a global transaction manager is mandatory to orchestrate the distributed transaction outcome.

All transactional resource adapters are registered by the global transaction manager, which decides when a resource is allowed to commit or rollback. The Java EE managed resources become accessible through JNDI (Java Naming and Directory Interface) or CDI (Contexts and Dependency Injection).

Spring provides a transaction management abstraction layer which can be configured to either use local transactions (JDBC or RESOURCE_LOCAL[8] JPA) or global transactions through a stand-alone JTA transaction manager. The dependency injection mechanism auto-wires managed resources into Spring beans.

1.6.1.1 Two-phase commit

JTA makes use of the two-phase commit (2PC) protocol to coordinate the atomic resource commitment in two steps: a *prepare* and a *commit* phase.

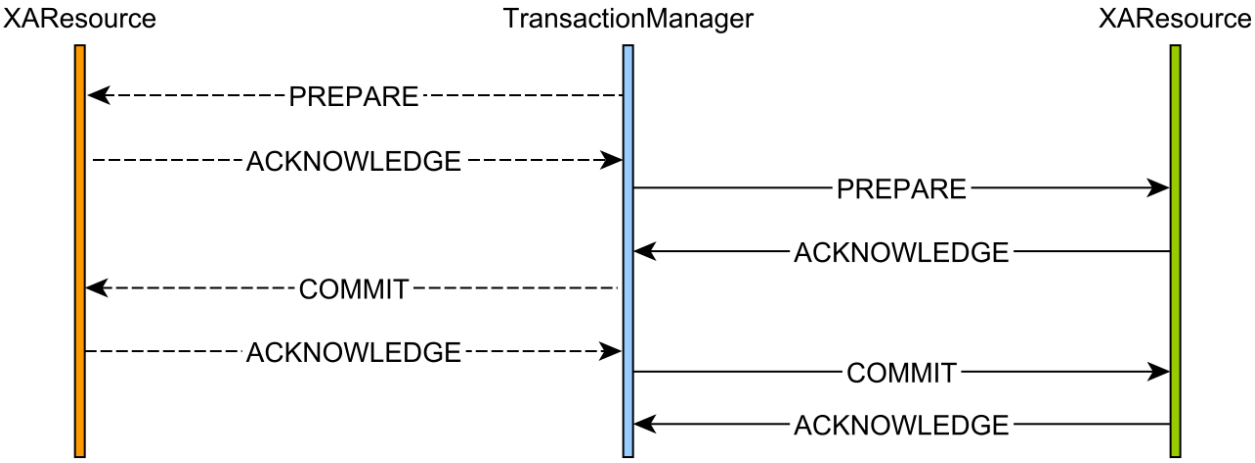

Figure 6.12: Two-phase commit protocol

In the former phase, a resource manager takes all the necessary actions to prepare the transaction for the upcoming commit. Only if all resource managers successfully acknowledge the preparation step, the transaction manager proceeds with the commit phase. If one resource does not acknowledge the prepare phase, the transaction manager will proceed to roll back all current participants.

If all resource managers acknowledge the commit phase, the global transaction will end successfully.

[8] http://docs.oracle.com/javaee/7/api/javax/persistence/spi/PersistenceUnitTransactionType.html#RESOURCE_LOCAL

If one resource fails to commit (or times out), the transaction manager will have to retry this operation in a background thread until it either succeeds or reports the incident for manual intervention.

The one-phase commit (1PC) optimization

Because Java EE uses JTA transactions exclusively, the extra coordination overhead of the additional database roundtrip may hurt performance in a high-throughput application environment. When a transaction enlists only one resource adapter (designating a single resource manager), the transaction manager can skip the prepare phase, and either execute the commit or the rollback phase. With this optimization, the distributed transaction behaves similarly to how JDBC `Connection(s)` manage local transactions.

The `XAResource.commit(Xid xid, boolean onePhase`[a] method takes a `boolean` flag, which the transaction manager sets to *true* to hint the associated resource adapter to initiate the 1PC optimization.

[a] https://docs.oracle.com/javaee/7/api/javax/transaction/xa/XAResource.html#commit-javax.transaction.xa.Xid-boolean-

1.6.2 Declarative transactions

Transaction boundaries are usually associated with a Service layer, which uses one or more DAO to fulfill the business logic. The transaction *propagates* from one component to the other within the service-layer transaction boundaries.

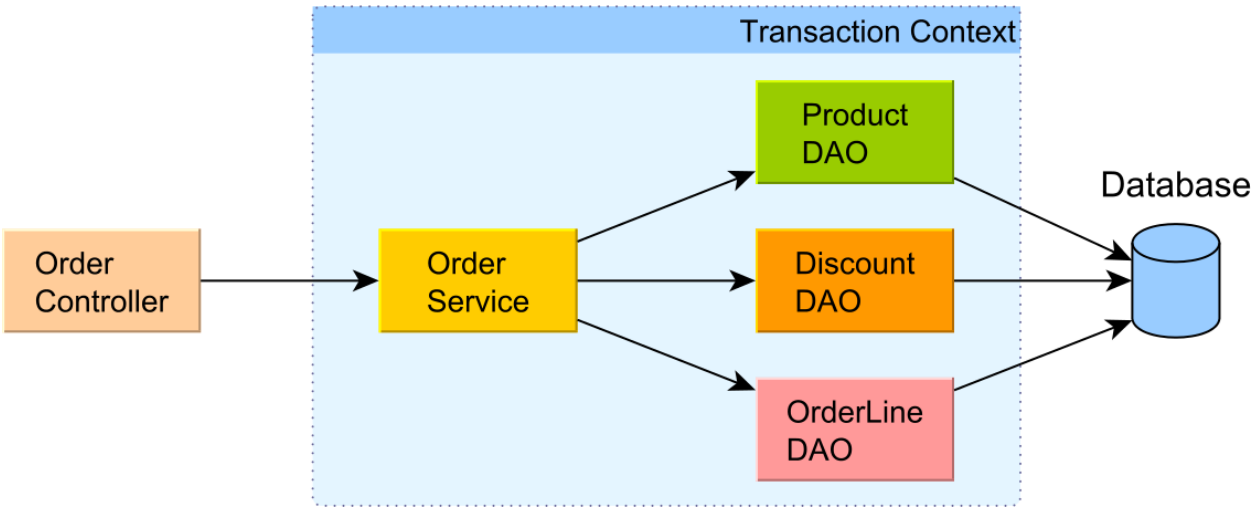

Figure 6.13: Transaction boundaries

The declarative transaction model is supported by both Java EE and Spring. Transaction boundaries are controlled through similar propagation strategies, which define how boundaries are inherited or disrupted at the borderline between the outermost component (in the current call stack) and the current one (waiting to be invoked).

Propagation

To configure the transaction propagation strategy for EJB components, Java EE defines the `@TransactionAttribute`[a] annotation. Since Java EE 7, even non-EJB components can now be enrolled in a transactional context if they are augmented with the `@Transactional`[b] annotation.

In Spring, transaction propagation (like any other transaction properties) is configurable via the `@Transactional`[c] annotation.

[a] http://docs.oracle.com/javaee/7/api/javax/ejb/TransactionAttribute.html
[b] http://docs.oracle.com/javaee/7/api/javax/transaction/Transactional.html
[c] https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/transaction/annotation/Transactional.html#propagation--

Table 6.6: Transaction propagation strategies

Propagation	Java EE	Spring	Description
REQUIRED	Yes	Yes	This is the default propagation strategy, and it only starts a transaction unless the current thread is not already associated with a transaction context
REQUIRES_NEW	Yes	Yes	Any currently running transaction context is suspended and replaced by a new transaction
SUPPORTS	Yes	Yes	If the current thread already runs inside a transaction, this method will use it. Otherwise, it executes outside of a transaction context
NOT_SUPPORTED	Yes	Yes	Any currently running transaction context is suspended, and the current method is run outside of a transaction context
MANDATORY	Yes	Yes	The current method runs only if the current thread is already associated with a transaction context
NESTED	No	Yes	The current method is executed within a nested transaction if the current thread is already associated with a transaction. Otherwise, a new transaction is started.
NEVER	Yes	Yes	The current method must always run outside of a transaction context, and, if the current thread is associated with a transaction, an exception will be thrown.

Declarative exception handling

Since the transaction logic wraps around the underlying service and data access logic call chain, the exception handling must also be configured declaratively. By default, both Java EE and Spring roll back on system exceptions (any `RuntimeException`) and commit on application exceptions (checked exceptions).

In Java EE, the rollback policy can be customized using the `@ApplicationException`[a] annotation.

Spring allows each transaction to customize the rolling back policy[b] by listing the exception types triggering a transaction failure.

[a] http://docs.oracle.com/javaee/7/api/javax/ejb/ApplicationException.html
[b] https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/transaction/annotation/Transactional.html#rollbackFor--

Declarative read-only transactions

Java EE does not support read-only transactions to be marked declaratively.

Spring offers the transactional read-only attribute[a], which can propagate to the underlying JPA provider (to optimize the `EntityManager` flushing mechanism) and to the current associated JDBC `Connection`.

[a] https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/transaction/annotation/Transactional.html#readOnly--

Declarative isolation levels

The Java EE does not offer support for configurable isolation levels, so it is up to the underlying `DataSource` to define it for all database connections.

Spring supports transaction-level isolation levels[a] when using the `JpaTransactionManager`[b]. For JTA transactions, the `JtaTransactionManager`[c] follows the Java EE standard and disallows overriding the default isolation level. As a workaround, the Spring framework provides extension points, so the application developer can customize the default behavior and implement a mechanism to set isolation levels on a transaction basis.

[a] https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/transaction/annotation/Transactional.html#isolation--
[b] https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/orm/jpa/JpaTransactionManager.html
[c] https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/transaction/jta/JtaTransactionManager.html

1.7 Application-level transactions

So far, the book focused on database transactions to enforce ACID properties. However, from the application perspective, a business workflow might span over multiple physical database transactions, in which case the database ACID guarantees are not sufficient anymore. A logical transaction may be composed of multiple web requests, including user think time, for which reason it can be visualized as a *long conversation*.

In the following example, *Alice* and a background process are concurrently modifying the same database record.

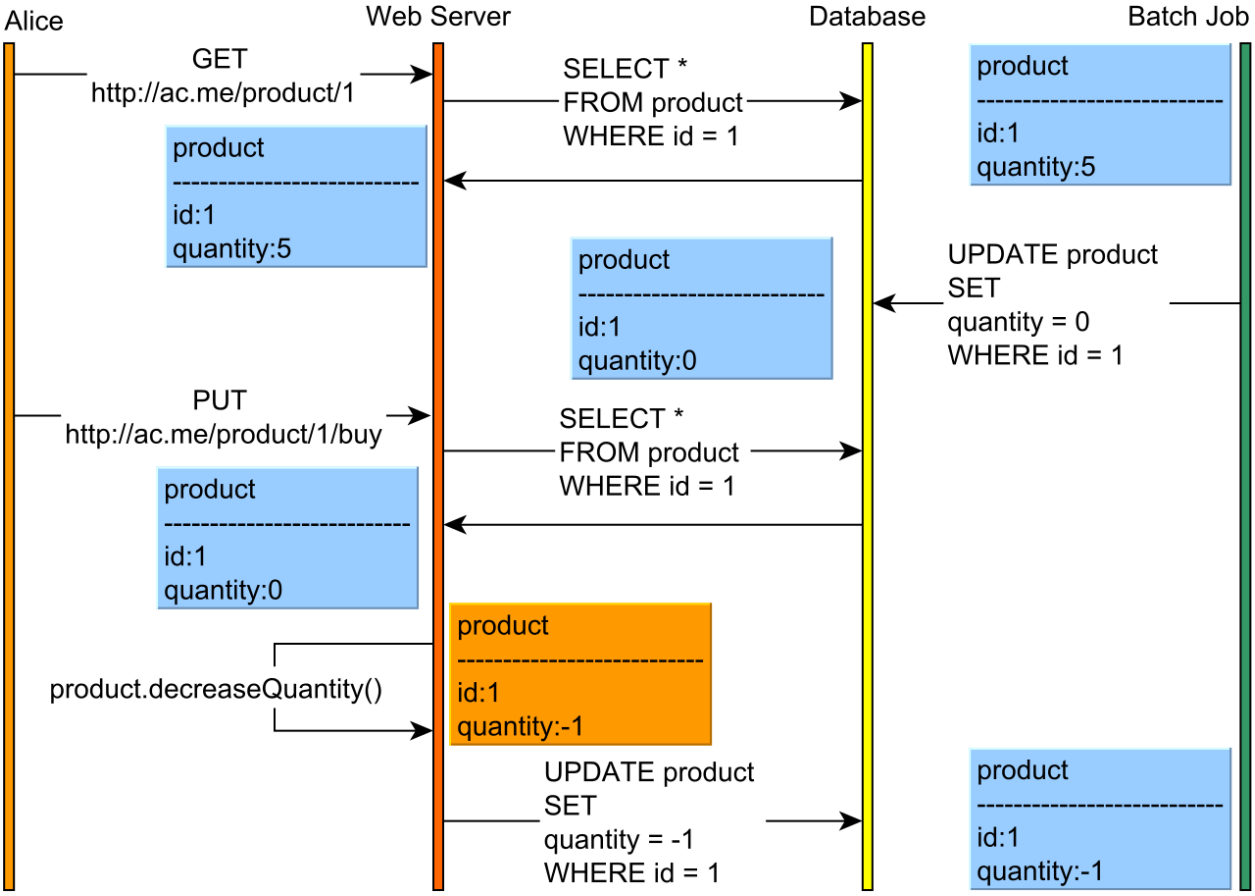

Figure 6.14: Stateless conversation losing updates

Because *Alice* logical transaction encloses two separate web requests, each one associated with a separate database transaction, without an additional concurrency control mechanism, even the strongest isolation level cannot prevent the lost update phenomena.

Spanning a database transaction over multiple web requests is prohibitive since locks would be held during user think time, therefore hurting scalability. Even with MVCC, the cost of maintaining previous versions (that can lead to a large version graph) can escalate and affect both performance and concurrency. Application-level transactions require application-level concurrency control mechanisms.

HTTP is stateless by nature and, for very good reasons, stateless applications are easier to scale than stateful ones. However, application-level transactions cannot be stateless, as otherwise newer requests would not continue from where the previous request was left. Preserving state across multiple web requests allows building a conversational context, providing application-level repeatable reads guarantees.

In the next diagram, *Alice* uses a stateful conversational context, but, in the absence of a record versioning system, it is still possible to lose updates.

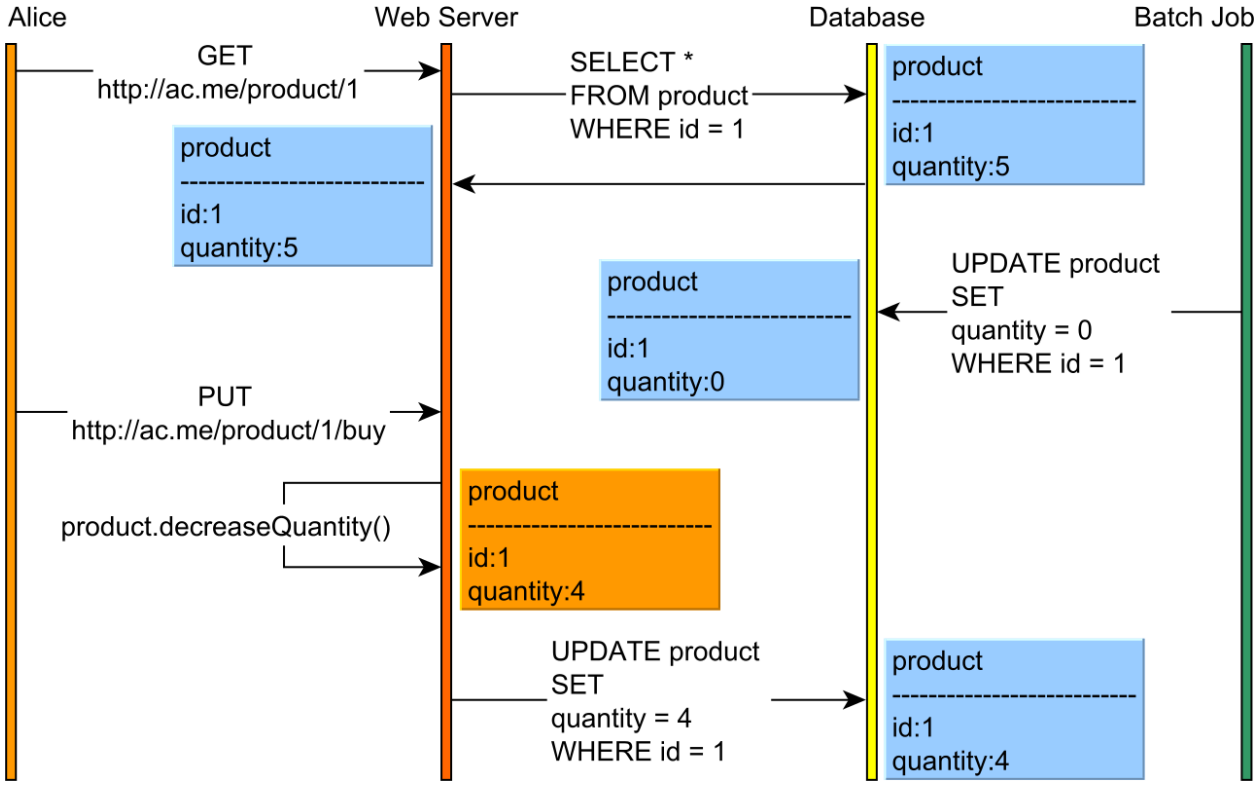

Figure 6.15: Stateful conversation losing updates

Without *Alice* to notice, the batch process resets the product quantity. Thinking the product version has not changed, *Alice* attempts to purchase one item which decreases the previous product quantity by one. In the end, *Alice* has simply overwritten the batch processor modification, and data integrity has been compromised.

So the application-level repeatable reads are not self-sufficient (this argument is true for database isolation levels as well). To prevent lost updates, a concurrency control mechanism becomes mandatory.

1.7.1 Pessimistic and optimistic locking

Isolation levels entail implicit locking, whether it involves physical locks (like 2PL) or data anomaly detection (MVCC). To coordinate state changes, application-level concurrency control makes use of explicit locking, which comes in two flavors: pessimistic and optimistic locking.

1.7.1.1 Pessimistic locking

As previously explained, most database systems already offer the possibility of manually requesting shared or exclusive locks. This concurrency control is said to be *pessimistic* because it assumes that conflicts are bound to happen, and so they must be prevented accordingly.

> As locks can be released in a timely fashion, exclusive locking is appropriate during the last database transaction of a given long conversation. This way, the application can guarantee that, once locks are acquired, no other transaction can interfere with the currently locked resources.
>
> 
>
> Acquiring locks on critical records can prevent non-repeatable reads, lost updates, as well as read and write skew phenomena.

1.7.1.2 Optimistic locking

Undoubtedly a misnomer (albeit rather widespread), optimistic locking does not incur any locking at all. A much better name would be optimistic concurrency control since it uses a totally different approach to managing conflicts than pessimistic locking.

MVCC is an optimistic concurrency control strategy since it assumes that contention is unlikely to happen, and so it does not rely on locking for controlling access to shared resources. The optimistic concurrency mechanisms detect anomalies and resort to aborting transactions whose invariants no longer hold.

While the database knows exactly which row versions have been issued in a given time interval, the application is left to maintaining a *happens-before* event ordering. Each database row must have an associated version, which is locally incremented by the logical transaction. Every modifying SQL statement (update or delete) uses the previously loaded version as an assumption that the row has not been changed in the meanwhile.

Because even the lowest isolation level can prevent write-write conflicts, only one transaction is allowed to update a row version at any given time. Since the database already offers monotonic updates, the row versions can also be incremented monotonically, and the application can detect when an updating record has become stale.

The optimistic locking concurrency algorithm looks like this:

- When a client reads a particular row, its version comes along with the other fields.
- Upon updating a row, the client filters the current record by the version it has previously loaded.

```
UPDATE product
SET (quantity, version) = (4, 2)
WHERE id = 1 AND version = 1
```

- If the statement update count is zero, the version was incremented in the meanwhile, and the current transaction now operates on a stale record version.

The previous example can be adjusted to take advantage of this optimistic concurrency control mechanism. This time, the product is versioned, and both the web application and the batch processor data access logic are using the row versions to coordinate the *happens-before* update ordering.

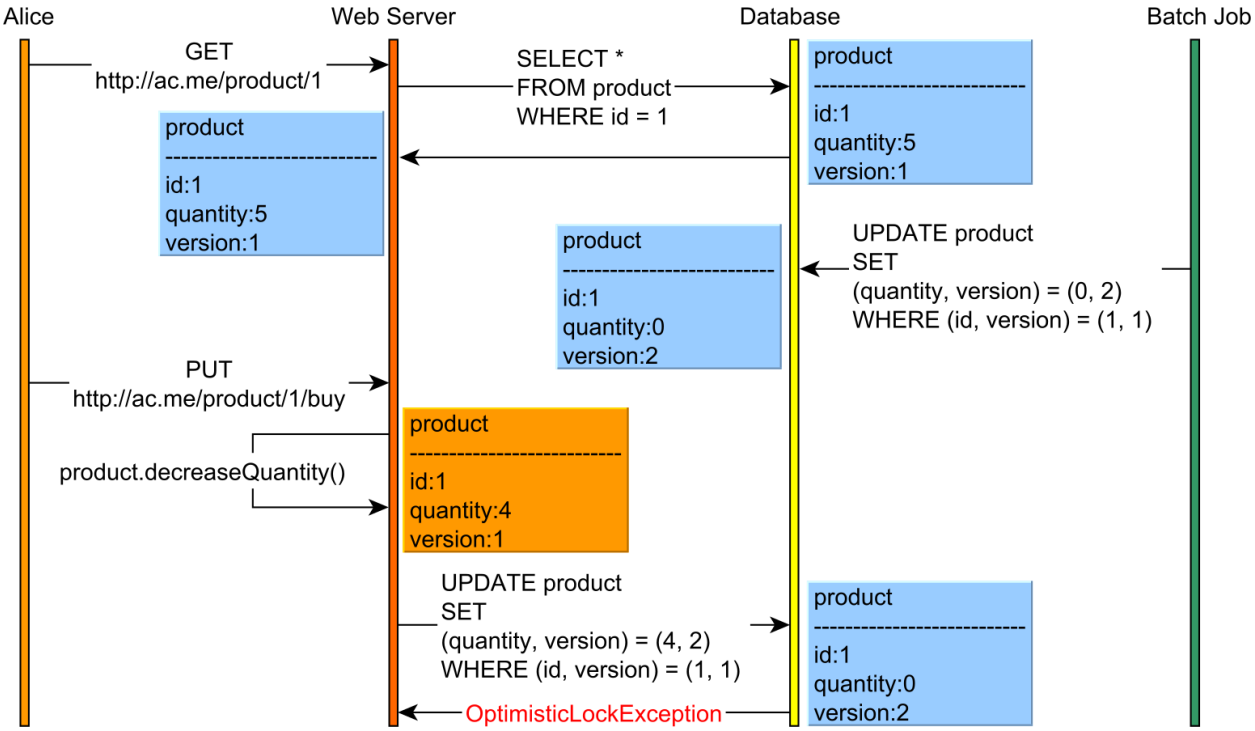

Figure 6.16: Stateful conversation preventing lost updates

Both *Alice* and the batch processor try to increment the product version optimistically. The batch processor can successfully update the product quantity since the SQL statement filtering criteria matches the actual database record version.

When *Alice* tries to update the product, the database returns a zero update count, and, this way, she is notified about the concurrent update that happened in the meanwhile.

The lost update can be prevented because the application can abort the current transaction when being notified of the stale record versions.