Spring Data - II

Teaching Faculty: Umur INAN

Prepared by Umur INAN

Parent-Child operations (Cascade Types)

- PERSIST
  - Cascading calls to EntityManager.persist() - persists children.
- MERGE
  - Cascading calls to EntityManager.merge() - updates children.
- ALL
  - Shortcut for cascade={PERSIST, MERGE, REMOVE, REFRESH}

Hibernate FETCH Strategies

- Select
- Join
- Subselect
- Batch

Hibernate FETCH Strategy -- SELECT

- Default
- N+1 Fetches
- a second SELECT [per parent N] is used to retrieve the associated collection.
- @Fetch(FetchMode.SELECT)
- To Be Avoided

Hibernate FETCH Strategy -- JOIN

- associated collections are retrieved in the same SELECT.
- uses an OUTER JOIN.
- 1 Fetch
- EAGER
- @Fetch(FetchMode.JOIN)
- Cartesian – need to watch collection sizes; can be useful strategy.

Hibernate FETCH Strategy -- JOIN

- ALWAYS Causes an EAGER fetch of the child collections.
- This is because the characteristic of a Join is ONE fetch Parent & Child TOGETHER
- This can only be accomplished by loading the child collection when the parent is fetched [~= EAGER fetch]
- It can be implemented Manually to use LAZY initialization.

Hibernate FETCH Strategy -- subselect

- a second SELECT is used to retrieve the associated collections for all entities retrieved in a previous query or fetch
- 2 Fetches
  - ORM will do ONE Fetch for All Parents
  - ORM will do ONE Fetch for All child collections

Hibernate FETCH Strategy -- SUBSELECT

- @Fetch(FetchMode.SUBSELECT)
- depends on the "parent" query. If parent Query is complex, it could have performance impacts.
- If fetch=FetchType.LAZY need to "hydrate" children

Hibernate FETCH Strategy -- BATCH

- Optimization of Select Fetching
- Associated collections are fetched according to declared Batch Size(N/Batch Size) + 1
- @BatchSize(size=n)
- Batch fetching is often called a blind-guess optimization
- # of Fetches “unknown“ UNLESS size of parent is constant

N+1 PROBLEM

- Prepare demo for n+1 problem

INHERITANCE

- Single Table
- Joined Tables [Table Per Subclass]
- Table per Class

Inheritance - single table

- Contains all columns for Super Class & ALL Sub Classes
- De-normalized schema
- Efficient queries
- Difficult to maintain as the number of columns increase.
- Fast polymorphic queries

Inheritance - joined table

- Table per Subclass
- Normalized schema
- Similar to OO classes
- Less efficient queries.
- Effective if hierarchy isn't too deep.
- Good if following GoF Patterns

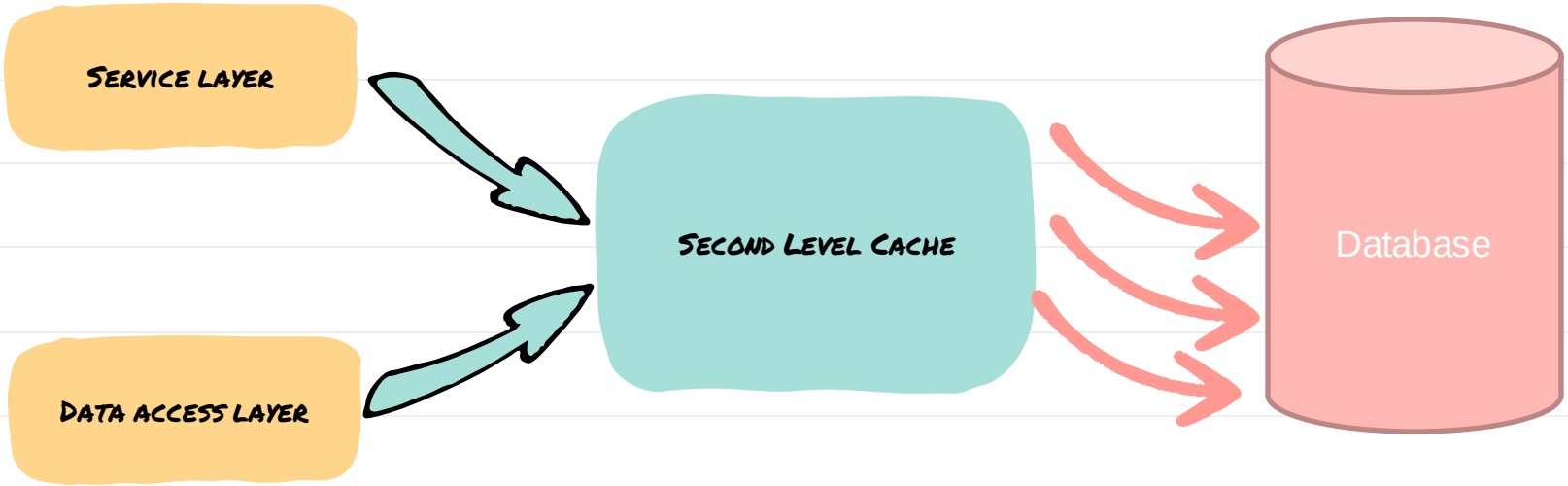
Inheritance – table Per class

- Super Class is replicated in each subclass table.
- Uses UNION instead of JOIN.
- All needed columns in each table.

Second level cache

- A local store of entity data managed by the persistence provider to improve application performance.
- Improves performance by avoiding expensive database calls.
- Keeps the entity data local to the application.
- Available across all users [Application wide].
- Complements First level cache.

Level 2 cache



Query for entity

- Check First level Cache
- If Found:
  - Return Entity
- If not Found:
  - Check Second level Cache
  - If Found:
    - Update First Level Cache
    - Return Entity
  - If not Found:
    - Execute DB Query
    - Update Second Level Cache
    - Update First Level Cache
    - Return Entity

Cache Concurrency Strategy -- read-write

- Read-Mostly data where it is critical to prevent stale data in concurrent transactions, in the rare case of an update.
- Supports Repeatable Read Isolation.
- Allows phantom reads.

Cache Concurrency Strategy -- nonstrict-read-write

- No guarantee of consistency between the cache and the database.
- Use this strategy if data hardly ever changes and a small likelihood of stale data is not of critical concern.

Cache Concurrency Strategy -- read-only

- Suitable for data which never changes.
- Use it for reference data only.
- Simplest and optimal performing strategy.

Second Level Cache Decision

- Make sure the fetching strategy is properly designed.
- Remove N+1 query problems.
- Employ indexes.
- Investigate data base solutions.

If performance is still an issue, THEN consider a second level cache.

ACID Database properties

- ATOMIC
  - The transaction is considered a single unit, either the entire transaction completes, or the entire transaction fails.

ACID Database properties

- CONSISTENT
  - A transaction transforms the database from one consistent state to another consistent state.

ACID Database properties

- ISOLATED
  - Data inside a transaction cannot be changed by another concurrent processes until the transaction has been committed.

ACID Database properties

- DURABLE
  - Once committed, the changes made by a transaction are persistent

Isolation in a Relational Database

- The challenge is to maximize concurrent transactions, while maintaining consistency.

- The shorter the lock acquisition interval, the more requests a database can process.

Spring Isolation Levels

- SERIALIZABLE:
  - NO dirty, non-repeatable OR phantom reads
- REPEATABLE READ
  - NO dirty OR non-repeatable reads
- READ COMMITTED
  - NO dirty reads
- READ UNCOMMITTED
  - ANYTHING Goes

Most databases default to READ COMMITTED

Spring Isolation Levels

Level	Dirty reads	Non-repeatable reads	Phantom reads
READ_UNCOMITTED	Yes	Yes	Yes
READ_COMITTED	No	Yes	Yes
REPEATABLE_READ	No	No	Yes
SERIALIZABLE	No	No	No

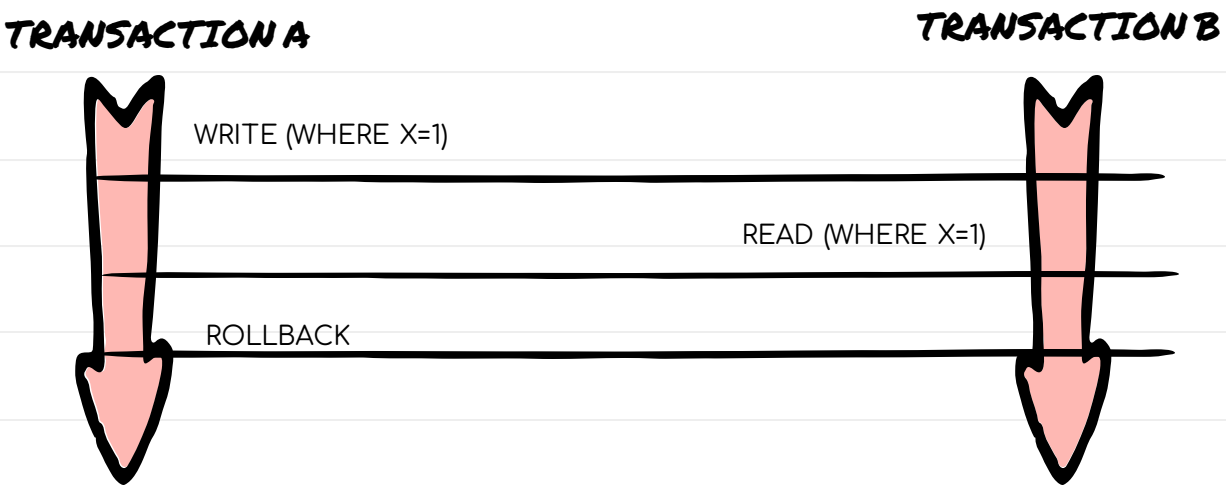
Transaction Read Phenomena

- Dirty Read:
  - Transaction B is allowed to read data from a row that has been modified by Transaction A and not yet committed If Transaction A rolls back, Transaction B has “bad” data.
- Non-repeatable read
  - Transaction A reads a row twice. Transaction B modifies the row between reads. Transaction A gets inconsistent data.
- Phantom read
  - Transaction A fetches a collection twice. Transaction B modifies the collection between fetches. Transaction A gets inconsistent data.

DIRTY READ

RECORD IN TX B IS NOW DIRTY

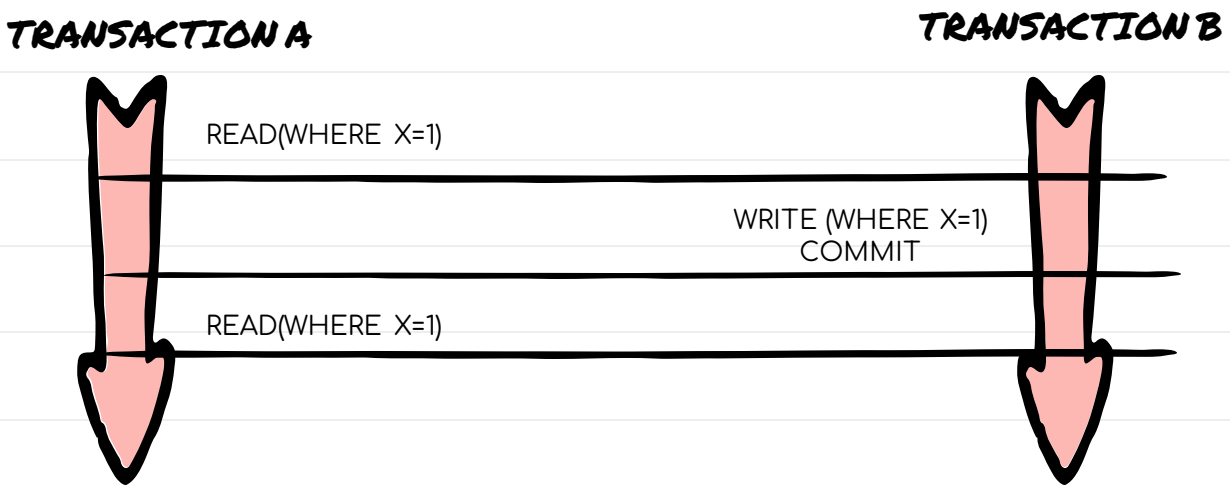
- Transaction B is allowed to read data from a row that has been modified by Transaction A and not yet committed.
- “x” is entity Primary key



NON-REPEATABLE READ

Tx A might get a record with different values between reads.

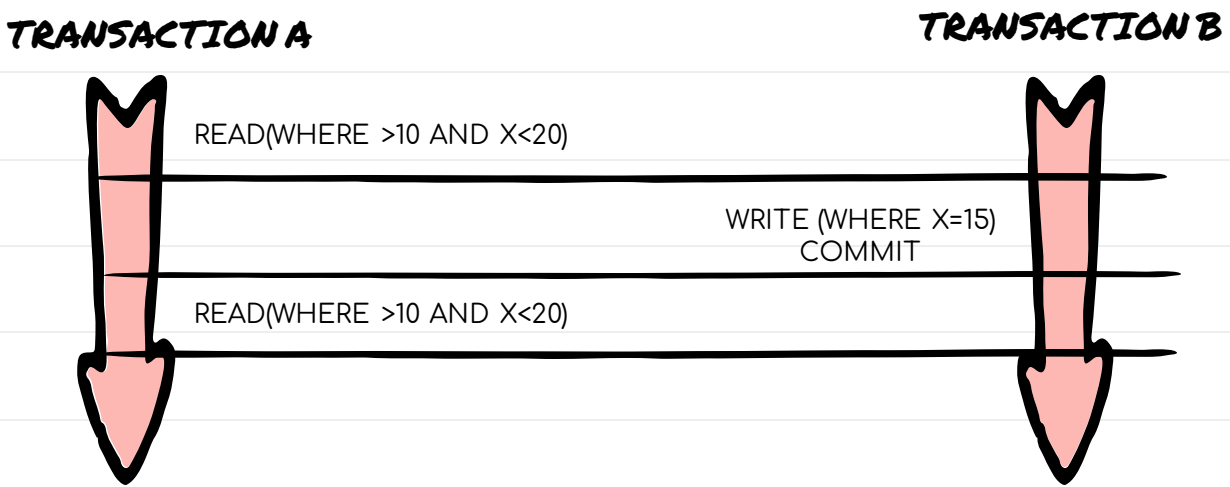
- Transaction A reads a row twice. Transaction B modifies the row between reads. Transaction A gets inconsistent data
- “x” is entity Primary key



PHANTOM READ

Results fetched by Tx A may be different in both reads.

- Transaction A fetches a collection twice. Transaction B modifies the collection between fetches.
- Transaction A gets inconsistent data.



Lock Mode for Data Consistency

- Locking refers to actions taken to prevent data in a relational database from changing between the time it is read and the time that it is used.

Optimistic Lock

- Concurrent transactions can complete without affecting each other.

- Transactions do not need to lock the data resources that they affect.

Pessimistic Lock

- Concurrent transactions will conflict with each other
- Transactions require that data is locked when read and unlocked at commit.

Version-Based Optimistic Concurrency

- High-volume systems.
- No Connection maintained to the Database.
- Effective in Read-Often Write-Sometimes scenario.

- Uses read committed isolation level.

Version-Based Optimistic Concurrency

- Optimistic concurrency assumes that update conflicts generally don't occur.
  - Keeps version #s so that it knows when they do
  - Guarantees best performance and scalability
  - The default way to deal with concurrency
  - There is no locking anywhere
  - It works well with very long conversations, including those that span multiple transactions

Version-Based Optimistic Concurrency

- First commit wins instead of last commit wins.
  - An exception is thrown if a conflict would occur:
    - ObjectOptimisticLockingFailureException

Main points

- Spring provides a Transactional capability for ORM applications.

- The mechanism of transcending allows the individual to tap into Transcendental Consciousness and enlivens its qualities in activity.