Module 08: Hibernate Concurrency

CS544: Enterprise Architecture

Transactions

- A Transaction is a unit of work that is:
  - **ATOMIC:** The transaction is considered a single unit, either the entire transaction completes, or the entire transaction fails.
  - **CONSISTENT:** A transaction transforms the database from one consistent state to another consistent state
  - **ISOLATED:** Data inside a transaction can not be changed by another concurrent processes until the transaction has been committed
  - **DURABLE:** Once committed, the changes made by a transaction are persistent

Hibernate Concurrency:

ISOLATION LEVELS

Isolation Levels

- Proper full isolation is expensive to produce in a multi user environment
  - Isolation is often relaxed to increase db speed
  - ANSI SQL defines four isolation levels

Read Uncommitted, Read Committed, Repeatable Read, and Serializable

Weaker, faster to Stronger, slower

- Most dbs default to Read Committed isolation
  - Only Serializable fully isolates a transaction from concurrency issues

Changing the Default Level

- You can raise the default isolation to a higher level to avoid some of these problems
  - Everything will be slower, less scalable
  - Even for transactions that don't need it
  - Recommend using optimistic concurrency instead

```
<hibernate-configuration>
  <session-factory>
    <!-- HSQL DB running on localhost -->
    <property name="connection.url">jdbc:hsqldb:hsql://localhost/trainingdb</property>
    <property name="connection.driver_class">org.hsqldb.jdbcDriver</property>
    <property name="connection.username">sa</property>
    <property name="connection.password"></property>
    <property name="dialect">org.hibernate.dialect.HSQLDialect</property>
    <property name="connection.isolation">8</property>

    ...
</session-factory>
</hibernate-configuration>
```

Specify the default isolation

1 – Read Uncommitted
2 – Read Committed
4 – Repeatable Read
8 – Serializable Isolation

Hibernate Concurrency:

OPTIMISTIC CONCURRENCY

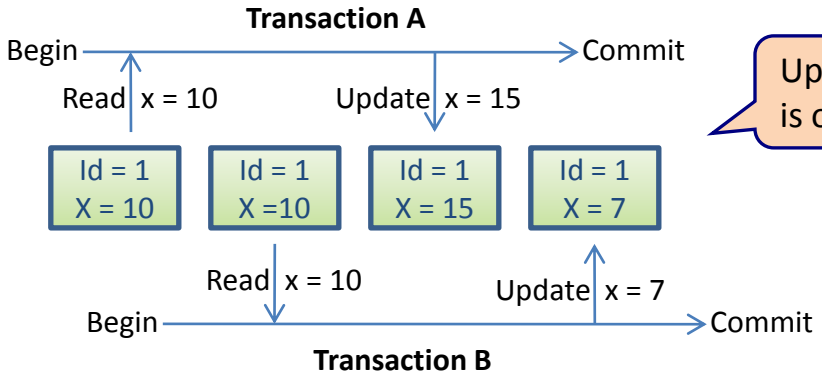
Optimistic Concurrency

- Optimistic concurrency assumes that lost update conflicts generally don't occur
  - But keeps versions# so that it knows when they do
  - Uses read committed transaction level
  - Guarantees best performance and scalability
  - The default way to deal with concurrency

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

Last Commit Wins (Lost Update)

- Normally the Read Committed level allows:



Update made by Transaction A is overwritten by Transaction B

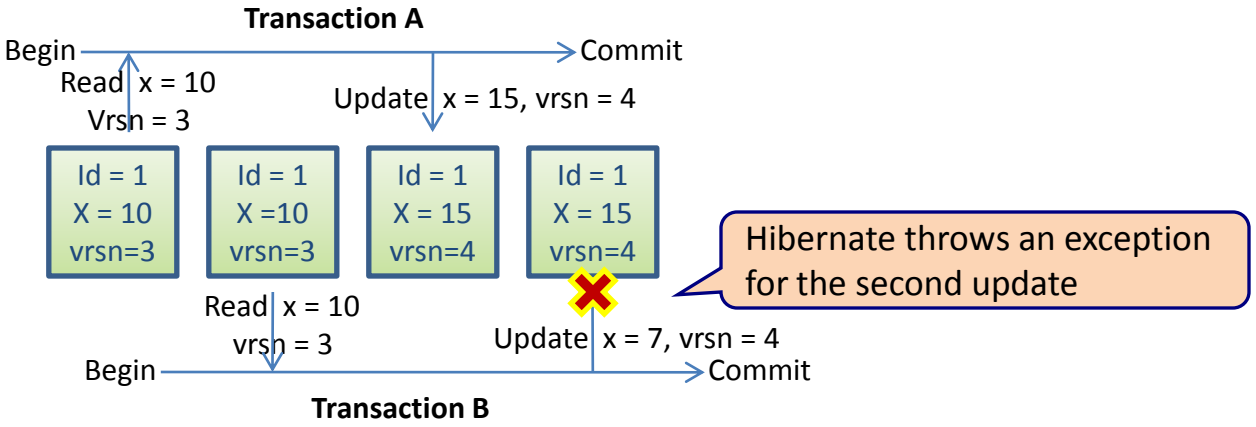
  - Transactions A and B read Id = 1, X = 10
  - Transaction A first increments X by 5 setting X = 15
  - Transaction B next decrements X by 3 and sets X = 7

believing X is still 10

- The update made by A is permanently lost, and neither A nor B is aware that it happened

First Commit Wins – Versioning

- Optimistic concurrency uses an additional version column to tracks updates



- Update Fails due to the version check
  - UPDATE table SET x = 15, vrsn = 4 WHERE id = 1 AND vrsn = 3
  - If the version has changed, the update is not executed
  - Hibernate throws an exception when the update fails

StaleObjectStateException

- When a version conflict occurs Hibernate throws a StaleObjectStateException
  - Catching this exception allows you to notify the user about the conflict
  - The user can then reload the data and apply their updates against the latest data

```
org.hibernate.StaleObjectStateException: Row was updated
or deleted by another transaction (or unsaved-value
mapping was incorrect): [optimistic.nocolumn.Customer#1]

...
```

Merging Conflicts

- Or you can create a conflict merging page
  - Showing the updates the other transaction made
  - Showing the updates the user wanted to make
  - Allowing the user to easily merge any differences

- This is the ideal solution for the user
  - They may not remember the details of their changes, or may get upset about re-entering them
  - Requires significantly more programming effort
  - May not be necessary for the occasional conflict

Version Column

- The best way to enable versioning for a class is by using an additional version column
  - Should have no semantic value in the table
  - Should be updated by all applications using the db

```
@Entity
public class Customer {
  @Id
  @GeneratedValue
  private int id;
  private String firstname;
  private String lastname;

  @Version
  private int version;

  ...
```

Use the @Version annotation to specify the version column

```
<hibernate-mapping package="optimistic.version">
  <class name="Customer">
    <id name="id">
      <generator class="native" />
    </id>

    <version name="version" access="field" />

    <property name="firstname" />
    <property name="lastname" />
  </class>
</hibernate-mapping>
```

Field access so that we don't need getter / setter

With XML the <version> tag should be placed after <id> and before any <property> tags

Timestamp Column

- Alternately a timestamp column can be used
  - Timestamps are considered less secure
  - Timestamps may fit the business logic better
  - The class may already have a timestamp column

```
@Entity
public class Customer {
  @Id
  @GeneratedValue
  private int id;
  private String firstname;
  private String lastname;

  @Version
  private Date timestamp;

  ...
```

@Version on a Date or Calendar object creates a versioning timestamp

```
<hibernate-mapping package="optimistic.timestamp">
  <class name="Customer">
    <id name="id">
      <generator class="native" />
    </id>

    <timestamp name="timestamp" access="field" />

    <property name="firstname" />
    <property name="lastname" />
  </class>
</hibernate-mapping>
```

XML uses <timestamp> instead of <version>

Without a Column

- Lastly Hibernate can also attempt to detect version conflicts without an additional column
  - Checks if all the object attributes are still the same as when the row was initially retrieved
  - Does not work for detached objects

```
@Entity
@org.hibernate.annotations.Entity(
  optimisticLock=OptimisticLockType.ALL,
  dynamicUpdate=true
)
public class Customer {
  @Id
  @GeneratedValue
  private int id;
  private String firstname;
  private String lastname;

  ...
```

Requires Hibernate extension to specify OptimisticLockType.ALL and dynamicUpdate=true

```
<hibernate-mapping package="optimistic.nocolumn">
  <class name="Customer"
      optimistic-lock="all" dynamic-update="true">

    <id name="id">
      <generator class="native" />
    </id>
    <property name="firstname" />
    <property name="lastname" />
  </class>
</hibernate-mapping>
```

XML uses optimistic-lock and dynamic-update attributes on the <class> tag

Hibernate Concurrency:

APPLICATION TRANSACTIONS

Application Transactions

- Application Transactions are longer running conversations, and can be seen as
  - A Unit of Work from the User Perspective
  - Spanning two or more screens

- The user expects these units of work to be
  - **A**tomic, **C**onsistent, **I**solated, and **D**urable
  - Submitting data after each screen would not allow us to roll back the entire unit of work (not Atomic)
  - Nor should you use a single database transaction across multiple screens
    - Keeping locks open during user think time

Detached Objects

- The easiest way to manage application transactions is by using detached objects
  - in combination with optimistic concurrency

- Changes are kept in the detached objects
  - These are not re-attached until the last operation
  - The objects are re-attached and all changes are committed within a single db transaction
  - Optimistic concurrency throws and exception if isolation failed while the objects were detached

Checkout Example

- Persistent Shopping Cart:
  - Order and order items are saved in the session
  - Changes to the order are immediately saved to the db
    - Thereby also updating the order's version column

- Checkout has the following screens:
  1. Confirm the items you are checking out → order becomes detached
  2. Add payment details to order → detached, does not save to db
  3. Specify shipping details → detached, does not save to db
  4. Confirm entire order including shipping → reattach, save/update all

- When the order is confirmed:
  - Reconnect and commit entire order object tree
  - Checkout fails if order has been changed in db while detached
  - Checkout fails if payment does not process

Hibernate Concurrency:

PESSIMISTIC LOCKING

Pessimistic Locking

- For certain operations optimistic concurrency might not be enough
  - Stricter isolation might be required to prevent the unrepeatable reads problem
  - Hibernate can request explicit database level locks to provide increased isolation
  - These locks will be released on commit

```
Customer cust = (Customer)session.get(Customer.class, 1);
session.lock(cust, LockMode.UPGRADE);
```

```
Customer cust = (Customer)session.get(Customer.class, 1, LockMode.UPGRADE);
```

Session.get() also supports a LockMode argument, allowing you to combine loading and locking in one

Hibernate Lock Modes

Lock Mode	Description
NONE	Default Mode - Only access the db if the object isn't in cache
READ	Performs a version check for the object against the db
FORCE	Force a version increment of the object in the database
UPGRADE	Version checks the object, and if possible perform a database level lock using SELECT … FOR UPDATE
UPGRADE_NOWAIT	Same as UPGRADE, but using SELECT … FOR UPDATE NOWAIT - If the underlying database supports it
WRITE	Used internally by Hibernate when writing to the db. Not for use in applications

- On databases that don't support SQL locking UPGRADE and UPGRADE_NOWAIT fall back to READ

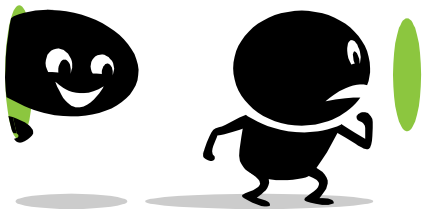
Hibernate Concurrency:

TRANSACTION REQUIREMENT

Transaction Requirement

- Many developers believe transactions are an optional part of database interactions
- In reality, there is no such thing as a database interaction without a transaction

- Most databases default to auto-commit mode
  - Wraps a transaction around each SQL statement
  - Effectively hiding the transaction from view



Auto Commit Mode

- Auto Commit Mode is good for SQL Console work, but bad for applications

  Console:

  - Work is often ad-hoc update / retrieval (no tx needed)
  - Having to add begin / commit would be more work

  Application:

  - More transactions means more overhead
  - Reduced isolation without transaction boundaries

- Hibernate disables auto commit by default
  - Thereby requiring *you* to specify when to commit

No Transaction?

New versions of Hibernate throw an exception

- If you don't specify a Hibernate transaction
  - A transaction will still be opened at the JDBC level
  - And Hibernate runs without auto-commit

```
Session session = sessionFactory.openSession();

session.save(new Customer("Frank", "Brown"));

session.close();
```

A connection is opened for session.save() which opens a JDBC transaction

- The TX is still open when the session is closed, which also closes the connection
  - Some dbs roll back on disconnect (HSQL, MS-SQL)
  - Other dbs commit on disconnect (MySQL, Oracle)

Omitting transaction code creates different results on different databases!

Active Learning

- How does optimistic concurrency work?
- What are application transactions?

Module Summary

- A transaction is a unit of work that is Atomic, Consistent, Isolated, and Durable (ACID)
- Full Isolation is expensive, most databases use lower isolation levels to increase performance
- Optimistic concurrency uses version checking to avoid data loss with concurrent updates
- Pessimistic locking can additionally be used to temporarily increase isolation
- Transactions are never optional – leaving off Hibernate TX code can create inconsistencies