Hibernate

Advanced concepts

First level cache

- Basic actions
- The request reaches the database server via the network.
- The database server processes the query in the query plan.
- Now the database server executes the processed query.
- Again, the database server returns the result to the querying application through the network.
- At last, the application processes the results.

- This process is repeated every time we request a database operation, even if it is for a simple or small query. It is always a costly transaction to hit the database for the same records multiple times.

contd

- To overcome this issue, the database uses a mechanism that stores the result of a query, which is executed repeatedly, and uses this result again when the data is requested using the same query.
- These operations are done on the database side. Hibernate provides an in-built caching mechanism known as the first-level cache (L1 cache).

Properties of L1 cache

- It is enabled by default. We cannot disable it even if we want to.
- The scope of the first-level cache is limited to a particular Session object only; the other Session objects cannot access it.
- All cached objects are destroyed once the session is closed.
- If we request for an object, hibernate returns the object from the cache only if the requested object is found in the cache; otherwise, a database call is initiated.
- We can use Session.evict(Object object) to remove single objects from the session cache.
- The Session.clear() method is used to clear all the cached objects from the session.

Test

- Test Level1 cache by repeatedly retrieving sets of data
- We find the query is executed only once for a certain data and subsequently retrieves from the cache
- Level1 cache cannot be disabled

Operations that lead to L1 caching

- Hibernate stores an object in the cache only if one of the following operations is completed:

- Save
- Update
- Get
- Load
- List
- methods used to remove a cached object from the session.

- There are two more methods that are used to remove a cached object:

- evict(Object object): This method removes a particular object from the session
- clear(): This method removes all the objects from the session

Working with a second level cache

- Scope of the second-level cache is SessionFactory, and we can use the cached objects across the different sessions that are created using this particular SessionFactory.
- Hibernate provides the option to either enable or disable the second-level cache.
- Hibernate provides a facility to change the cache provider, which means that we can provide any cache provider that supports integration with hibernate.
- Ehcache is used as the default cache provider by hibernate

Configuration

- Use the cache jars that come from the cache folder in hibernate distribution and place them in your classpath

Enabling L2 cache

- To enable the second-level cache, we need to add two new mappings in the configuration (CFG) file.

<property name="hibernate.cache.use_second_level_cache"> true</property>

<property name=*"cache.region.factory_class">org.hibernate.cache.ehcache.EhCacheRegionFactory*

</property>

The first property tag is used to enable the second-level cache. We must set the value of the hibernate.cache.use_second_level_cache property to true.

Another tag is used to provide a cache provider class, which is vendor-specific.

Adding caching to a pojo class

- @Entity
- @Table(name="employee")
- @Cache(usage=CacheConcurrencyStrategy.READ_ONLY)
- public class Employee {
- // fields and getters/setters
- }

- CacheConcurrencyStrategy.READ_ONLY: This strategy is suitable where the data never changes but is required frequently.
- CacheConcurrencyStrategy.NONSTRICT_READ_WRITE: This strategy is suitable for the applications that only rarely need to modify data.
- CacheConcurrencyStrategy.READ_WRITE: This strategy is suitable for the applications that regularly need to modify data.
- CacheConcurrencyStrategy.TRNSACTIONAL: The transactional cache strategy provides support to transactional cache providers such as JBoss TreeCache.

Test

- /* Line 1*/ Session session = sessionFactory.openSession();
- /* Line 2 */ Employee employee = (Employee) session.load(Employee.class, new Long(1));
- System.out.println(employee.toString());
- /* Line 4 */ session.close();
-
- /* Line 6 */ Session anotherSession = sessionFactory.openSession();
- /* Line 7 */ Employee employee_dummy = (Employee) anotherSession.load(Employee.class, new Long(1));
- System.out.println(employee_dummy.toString());
- /* Line 9 */ anotherSession.close();

2nd level caching flow

- When hibernate tries to load a particular entity, it first looks for the first-level cache of the current session.
- It is returned if the requested entity is present in the first-level cache.
- If this particular entity is not found in the first-level cache, it will look for the second-level cache.
- If the entity is found in the second-level cache, it's returned. Hibernate also stores this entity in the particular session, so there is no need to go to the second-level cache on the next request.
- If the entity is not found in the second-level cache, hibernate hits the database and stores it in both the first and second-level caches and then returns it.

Interceptors and Events

- in the hibernate persistent lifecycle, a particular object travels from state to state, from transient, persistent, to detached. During processing, it may commit or roll back before it reaches the last state. Sometimes, we need to perform some additional tasks such as cleanup, log or some operations on the object between different states of the persistent life cycle. To perform such activities, hibernate provides a useful and pluggable feature called interceptor.

- Interceptor, is used to intercept any operation. Interceptors apply hooks inside the logic. In hibernate, we have some built-in interceptors that help us intercept our logic.

- First of all, we need to create an interceptor class that extends EmptyInterceptor, which implements the org.hibernate.Interceptor interface and override the required methods :
- onSave, onLoad,onDelete,preFlush,postFlush

- onSave - The onSave method is called before the object is saved to the database
- preFlush -This method is called after commit is completed and just before flush is started.
- postFlush - This method is called after commit is completed and just before flush is completed

Configuration

Interceptor can be configured at a session level and at a session factory level.

At a Session Level - Pass an instance of Interceptor while opening a session.

hibernate 4 :

SessionFactory.withOptions() .interceptor(Interceptor()).openSession();

This way we can get a control at a session level and can enable interceptor for a desired sessions only.

At a Session Factory Level – Set an interceptor in a Configuration Object. Doing this , interceptor will be enabled for all the session for the session factory.

Like Configuration.setInterceptor(Interceptor)

Hibernate event system

- Hibernate comes with an event system (a concept of listeners and event) l and we can configure the events we want to track in the form of listeners. Most commonly used events are-

- Pre – Select: : these types of event are fired before executing any select statement.
- Post – Select: : these types of event are fired after executing any select statement.
- Pre – Insert: : these types of event are fired before execution of insert statement.
- Post – Insert: : these types of event are fired after execution of an insert statement.
- Pre – Update: : these types of event are fired before execution of update statements.
- Post – Update: : these types of event are fired after execution of any update statement.
- Pre – Delete: these types of event are fired before execution of any delete statement.
- Post – Delete: these types of event are fired after execution of any delete statement.

Implementing listeners

- Depending on the type of event we would need to implement the Listener interface available in org.hibernate.event package.
- Event is passed in the listener and we can get the entity by calling getEntity() method of listener.
- By default listener gets activated on all entities so we need to filter the entities using instaneof we are interested in.
- Interceptors can be added to the sesion factory

```
public static void registerListeners(SessionFactory sessionFactory) {
                        UpdateBookEventListener listener = new UpdateBookEventListener();
    EventListenerRegistry registry = ((SessionFactoryImpl)
sessionFactory).getServiceRegistry().getService(
    EventListenerRegistry.class);
    registry.getEventListenerGroup(EventType.PRE_UPDATE).appendListener(listener);
    registry.getEventListenerGroup(EventType.POST_UPDATE).appendListener(listener);
  }
```

Batchprocessing

- Sometimes, we need to save a large number of records in the database. Let's say we need 10,000 records in the database, and we use the basic approach of iterating to save the records.

```
Session session = SessionFactory.openSession();
Transaction tx = session.beginTransaction();
for ( int i=0; i<100000; i++ ) {
    Employee employee = new Employee(.....);
    session.save(employee);
}
tx.commit();
session.close();
```

Issues in existing

- Two known issues in the preceding method are as follows:

- Hibernate will try to save each object to the database one by one; this will be time consuming and may increase the load on the database and application as well
- The application may face OutOfMemoryException because hibernate saves all the new employee objects in the second-level cache
- To overcome these problems and to make the application faster, we need to use batch processing. Hibernate supports batch processing, which is the same as a JDBC batch processing.

batch

- First of all, the change in hibernate.cfg.xml is hibernate.jdbc.batch_size = 50, informing hibernate to create a batch of 50 for the batch operation.
- Another is hibernate.cache.use_second_level_cache = false, informing hibernate not to cache the object as we were doing a batch operation, and it is unnecessary to store the objects in the cache.

Code that flushes at a certain interval

```
Session session = sessionFactory.openSession();
Transaction transaction = session.beginTransaction();
for (int i = 0; i < 10000; i++) {
  Employee employee = new Employee();
  employee.setName("Name : " + String.valueOf(i));
  session.save(employee);
/* Line 7 */  if (i % 50 == 0) {
/* Line 8 */   session.flush();
/* Line 9*/   session.clear();
  }
}
transaction.commit();
session.close();
```

Contd.

- In the executable code, we looped 10,000 times and saved the records using the session.save(…) method. In Line 7, we checked whether the value of the i variable was equal to a multiple of 50 and then flushed and cleared the session.

- The Session.flush() method was used to persist a record and sent to the database. You cannot actually see it from the database or using another session/thread because this record is not committed yet. Once a transaction is committed, the records are available for the other session/thread.

- The Session.clear() method clears all the cached records from the session and releases the memory.

Transactions and concurrency

- The concept of transactions is inherited from database management systems. By definition, a transaction must be atomic, consistent, isolated, and durable (ACID):

- Atomicity means that if one step fails, the whole unit of work fails.
- Consistency means that the transaction works on a set of data that is consistent before and after the transaction. The data must be clean after the transaction. From a database perspective, the clean and consistent state is maintained by integrity constraints. From an application's perspective, the consistent state is maintained by the business rules.
- Isolation means that one transaction isn't visible to other transactions. Isolation makes sure that the execution of a transaction doesn't affect other transactions.
- Durability means that when data has been persisted, it isn't lost.

Transaction and concurrency

- In an application, you need to identify when a transaction begins and when it ends. The starting and ending points of a transaction are called *transaction boundaries*, and the technique of identifying them in an application is called *transaction demarcation*
- You can set the transaction boundaries either programmatically or declaratively.

- Multiuser systems such as databases need to implement concurrency control to maintain data integrity. There are two main approaches to concurrency control:

- Optimistic: Involves some kind of versioning to achieve control
- Pessimistic: Uses a locking mechanism to obtain control

transaction

- You can set the transaction boundaries either programmatically or declaratively.

- Multiuser systems such as databases need to implement concurrency control to maintain data integrity. There are two main approaches to concurrency control:

- Optimistic: Involves some kind of versioning to achieve control
- Pessimistic: Uses a locking mechanism to obtain control

Programmatic tx in standalone

- Using Programmatic Transactions in a Stand-alone Java Application

```
public void testTransactionWithExceptionHandling() {
  Session session = SessionManager.openSession();
  Transaction tx = session.beginTransaction();
  try {     Book1 book1 = new Book1();
    book1.setTitle("The Dog Barker");
    session.persist(book1);
    tx.commit();
  } catch (Exception e) {
    tx.rollback();
  } finally {     session.close();   }
}
```

Transaction using JTA

- Requires connection pool configuration to use a certain tx implementation
- <hibernate-configuration>
- <session-factory name="book">
- <property name="hibernate.connection.datasource">local_derby</property>
- <property name="hibernate.transaction.factory_class">org.hibernate.transaction.JTATransactionFactory</property>
- <property name="hibernate.transaction.manager_lookup_class">org.hibernate.transaction.WeblogicTransactionManagerLookup</property>
- <property name="hibernate.dialect">org.hibernate.dialect.DerbyDialect</property>
- <property name="hibernate.show_sql">true</property>
- <property name="hibernate.cache.use_second_level_cache">false</property>
- <mapping resource="book.xml" />
- </session-factory>
- </hibernate-configuration>

- The User transaction is looked up using a jndi look up

```
public void saveBook(Book book)
  throws NotSupportedException, SystemException, NamingException, Exception {
  System.out.println("Enter DAO Impl");
  Session session = null;
  UserTransaction tx = (UserTransaction)new InitialContext()
    .lookup("java:comp/UserTransaction");
  try {
    SessionFactory factory = HibernateUtil.getSessionFactory();
    tx.begin();
    session = factory.openSession();
    session.saveOrUpdate(book);
    session.flush();
    tx.commit();
  }catch (Exception e) {
    tx.rollback();
  } finally {
    session.close();
  }
}
```

Note..

- Note that you explicitly call session.flush(). You need to do an explicit flush because Hibernate’s default implementation doesn’t flush the session. You can, however, override the default implementation by configuring
  the hibernate.transaction.flush_before_completionproperty. You can also configure the hibernate.transaction.auto_close_session property to avoid calling session.close() explicitly in every method.

Optimistic concurrency control

- Suppose that two transactions are trying to update a record in the database.
- The first transaction updates and commits successfully, but the second transaction tries to update and fails. So the transaction is rolled back.
- The problem is that the first update is lost. Or what if the second transaction successfully updates.
- The changes made by the first transaction are overwritten.

Isolation levels

- A *dirty read* occurs when one transaction reads changes made by another transaction that haven't yet been committed
- An *unrepeatable read* occurs when a transaction reads a record twice, and the record state is different between the first and the second read. This happens when another transaction updates the state of the record between the two reads
- A *phantom read* occurs when a transaction executes two identical queries, and the collection of rows returned by the second query is different from the first. It also happens when another transaction inserts records into or deletes records from the table between the two reads.

Isolation

- *Isolation*, which is one of the ACID properties, defines how and when changes made by one transaction are made visible to other transactions.

Isolation

- *Serializable:* Transactions are executed serially, one after the other. This isolation level allows a transaction to acquire read locks or write locks for the entire range of data that it affects. The Serializable isolation level prevents dirty reads, unrepeatable reads, and phantom reads, but it can cause scalability issues for an application.
- *Repeatable Read:* Read locks and write locks are acquired. This isolation level doesn’t permit dirty reads or unrepeatable reads. It also doesn’t acquire a range lock, which means it permits phantom reads. A read lock prevents any write locks from being acquired by other concurrent transaction. This level can still have some scalability issues.

- *Read Committed:* Read locks are sacquired and released immediately, and write locks are acquired and released at the end of the transaction. Dirty reads aren’t allowed in this isolation level, but unrepeatable reads and phantom reads are permitted. By using the combination of persistent context and versioning, you can achieve the Repeatable Read isolation level.
- *Read Uncommitted*: Changes made by one transaction are made visible to other transactions before they’re committed. All types of reads, including dirty reads, are permitted. This isolation level isn’t recommended for use. If a transaction’s uncommitted changes are rolled back, other concurrent transactions may be seriously affected.

Hibernate - isolation

- Hibernate uses the following values to set a particular isolation level:
- 8: Serializable isolation
- 4: Repeatable Read isolation
- 2: Read Committed isolation
- 1: Read Uncommitted isolation
- This setting is applicable only when the connection isn’t obtained from an application server. In this scenario, you need to change the isolation level in the application server configuration.

optimistic locking - Versioning

- 1. create a field with type Integer called version and annotate it with @version

```
 @Version
private Integer version;
```

2. Annotate the entity as follows:

```
@org.hibernate.annotations.Entity
 (dynamicInsert = true, dynamicUpdate = true,
                optimisticLock=org.hibernate.annotations.OptimisticLockType.ALL)
@Table        (name="BOOK")
public class Book {
```

Pessimistic control – use session.lock()

```
Session session = getSession();
Transaction tx = null;
tx = session.beginTransaction();
Book book = (Book)session.get(BookCh2.class, new Long(32769));
session.lock(book, LockMode.UPGRADE);
String name = (String) session.createQuery("select b.name from bk b where b.isbn = :isbn")
                .setParameter("isbn", book.getIsbn()).uniqueResult();
System.out.println("BOOk's Name- "+name);
tx.commit();
session.close();
```

Thank you