Prev Next

Chapter 14. Batch processing

A naive approach to inserting 100,000 rows in the database using Hibernate might look like this:

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

This would fall over with an OutOfMemoryException somewhere around the 50,000th row. That is because Hibernate caches all the newly inserted Customer instances in the session-level cache. In this chapter we will show you how to avoid this problem.

If you are undertaking batch processing you will need to enable the use of JDBC batching. This is absolutely essential if you want to achieve optimal performance. Set the JDBC batch size to a reasonable number (10-50, for example):

```
hibernate.jdbc.batch_size 20
```

Hibernate disables insert batching at the JDBC level transparently if you use an identity identifier generator.

You can also do this kind of work in a process where interaction with the second-level cache is completely disabled:

```
hibernate.cache.use_second_level_cache false
```

However, this is not absolutely necessary, since we can explicitly set the CacheMode to disable interaction with the second-level cache.

14.1. Batch inserts

When making new objects persistent flush() and then clear() the session regularly in order to control the size of the first-level cache.

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

for ( int i=0; i<100000; i++ ) {
```

```
    Customer customer = new Customer(.....);
    session.save(customer);
    if ( i % 20 == 0 ) { //20, same as the JDBC batch size
        //flush a batch of inserts and release memory:
        session.flush();
        session.clear();
    }
}

tx.commit();
session.close();
```

14.2. Batch updates

For retrieving and updating data, the same ideas apply. In addition, you need to use scroll() to take advantage of server-side cursors for queries that return many rows of data.

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

ScrollableResults customers = session.getNamedQuery("GetCustomers")
    .setCacheMode(CacheMode.IGNORE)
    .scroll(ScrollMode.FORWARD_ONLY);
int count=0;
while ( customers.next() ) {
    Customer customer = (Customer) customers.get(0);
    customer.updateStuff(...);
    if ( ++count % 20 == 0 ) {
        //flush a batch of updates and release memory:
        session.flush();
        session.clear();
    }
}

tx.commit();
session.close();
```

14.3. The StatelessSession interface

Alternatively, Hibernate provides a command-oriented API that can be used for streaming data to and from the database in the form of detached objects. A StatelessSession has no persistence context associated with it and does not provide many of the higher-level life cycle semantics. In particular, a stateless session does not implement a first-level cache nor interact with any second-level or query cache. It does not implement transactional write-behind or automatic dirty checking. Operations performed using a stateless session never cascade to associated instances. Collections are ignored by a stateless session. Operations performed via a stateless session bypass Hibernate's event model and interceptors. Due to the lack of a first-level cache, Stateless sessions are vulnerable to data aliasing effects. A stateless session is a lower-level abstraction that is much closer to the underlying JDBC.

```
StatelessSession session = sessionFactory.openStatelessSession();
Transaction tx = session.beginTransaction();

ScrollableResults customers = session.getNamedQuery("GetCustomers")
    .scroll(ScrollMode.FORWARD_ONLY);
while ( customers.next() ) {
    Customer customer = (Customer) customers.get(0);
    customer.updateStuff(...);
    session.update(customer);
}

tx.commit();
session.close();
```

In this code example, the Customer instances returned by the query are immediately detached. They are never associated with any persistence context.

The insert(), update() and delete() operations defined by the StatelessSession interface are considered to be direct database row-level operations. They result in the immediate execution of a SQL INSERT, UPDATE or DELETE respectively. They have different semantics to the save(), saveOrUpdate() and delete() operations defined by the Session interface.

14.4. DML-style operations

As already discussed, automatic and transparent object/relational mapping is concerned with the management of the object state. The object state is available in memory. This means that manipulating data directly in the database (using the SQL Data Manipulation Language (DML) the statements: INSERT, UPDATE, DELETE) will not affect in-memory state. However, Hibernate provides methods for bulk SQL-style DML statement execution that is performed through the Hibernate Query Language (HQL).

The pseudo-syntax for UPDATE and DELETE statements is: (UPDATE | DELETE) FROM? EntityName (WHERE where_conditions)?.

Some points to note:

- In the from-clause, the FROM keyword is optional
- There can only be a single entity named in the from-clause. It can, however, be aliased. If the entity name is aliased, then any property references must be qualified using that alias. If the entity name is not aliased, then it is illegal for any property references to be qualified.
- No joins, either implicit or explicit, can be specified in a bulk HQL query. Sub-queries can be used in the where-clause, where the subqueries themselves may contain joins.
- The where-clause is also optional.

As an example, to execute an HQL UPDATE, use the Query.executeUpdate() method. The method is named for those familiar with JDBC's PreparedStatement.executeUpdate():

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

String hqlUpdate = "update Customer c set c.name = :newName where c.name = :oldName";
// or String hqlUpdate = "update Customer set name = :newName where name = :oldName";
int updatedEntities = s.createQuery( hqlUpdate )
        .setString( "newName", newName )
        .setString( "oldName", oldName )
        .executeUpdate();
tx.commit();
session.close();
```

In keeping with the EJB3 specification, HQL UPDATE statements, by default, do not effect the version or the timestamp property values for the affected entities. However, you can force Hibernate to reset the version or timestamp property values through the use of a versioned update. This is achieved by adding the VERSIONED keyword after the UPDATE keyword.

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
String hqlVersionedUpdate = "update versioned Customer set name = :newName where name = :oldName";
int updatedEntities = s.createQuery( hqlUpdate )
        .setString( "newName", newName )
        .setString( "oldName", oldName )
        .executeUpdate();
tx.commit();
session.close();
```

Custom version types, org.hibernate.usertype.UserVersionType, are not allowed in conjunction with a update versioned statement.

To execute an HQL DELETE, use the same Query.executeUpdate() method:

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

String hqlDelete = "delete Customer c where c.name = :oldName";
// or String hqlDelete = "delete Customer where name = :oldName";
int deletedEntities = s.createQuery( hqlDelete )
        .setString( "oldName", oldName )
        .executeUpdate();
tx.commit();
session.close();
```

The int value returned by the Query.executeUpdate() method indicates the number of entities effected by the operation. This may or may not correlate to the number of rows effected in the database. An HQL bulk operation might result in multiple actual SQL statements being executed (for joined-subclass, for example). The returned number indicates the number of actual entities affected by the statement. Going back to the example of joined-subclass, a delete against one of the subclasses may actually result in deletes against not just the table to which that subclass is mapped, but also the "root" table and potentially joined-subclass tables further down the inheritance hierarchy.

The pseudo-syntax for INSERT statements is: INSERT INTO EntityName properties_list select_statement. Some points to note:

- Only the INSERT INTO ... SELECT ... form is supported; not the INSERT INTO ... VALUES ... form.

  The properties_list is analogous to the column specification in the SQL INSERT statement. For entities involved in mapped inheritance, only properties directly defined on that given class-level can be used in the properties_list. Superclass properties are not allowed and subclass properties do not make sense. In other words, INSERT statements are inherently non-polymorphic.
- select_statement can be any valid HQL select query, with the caveat that the return types must match the types expected by the insert. Currently, this is checked during query compilation rather than allowing the check to relegate to the database. This might, however, cause problems between Hibernate Types which are *equivalent* as opposed to *equal*. This might cause issues with mismatches between a property defined as a org.hibernate.type.DateType and a property defined as a org.hibernate.type.TimestampType, even though the database might not make a distinction or might be able to handle the conversion.
- For the id property, the insert statement gives you two options. You can either explicitly specify the id property in the properties_list, in which case its value is taken from the corresponding select expression, or omit it from the properties_list, in which case a generated value is used. This latter option is only available when using id generators that operate in the database; attempting to use this option with any "in memory" type generators will cause an exception during parsing. For the purposes of this discussion, in-database generators are considered to be org.hibernate.id.SequenceGenerator (and its subclasses) and any implementers of org.hibernate.id.PostInsertIdentifierGenerator. The most notable exception here is org.hibernate.id.TableHiLoGenerator, which cannot be used because it does not expose a selectable way to get its values.
- For properties mapped as either version or timestamp, the insert statement gives you two options. You can either specify the property in the properties_list, in which case its value is taken from the corresponding select expressions, or omit it from the properties_list, in which case the seed value defined by the org.hibernate.type.VersionType is used.

The following is an example of an HQL INSERT statement execution:

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

String hqlInsert = "insert into DelinquentAccount (id, name) select c.id, c.name from Customer c wher
int createdEntities = s.createQuery( hqlInsert )
        .executeUpdate();
tx.commit();
session.close();
```


Prev Top of page Next

Chapter 13. Interceptors and events Chapter 15. HQL: The Hibernate Query Language