Hibernate 4.2 ORM Performance

Docs

Official user guide:

- http://hibernate.org/orm/documentation/4.2/

API Doc:

- https://docs.jboss.org/hibernate/orm/4.2/javadocs/

Connection between the database and Hibernate

- Hibernate creates a bridge between OO and RDBMS
- Hibernate's performance tuning requires deep SQL and RDBMS knowledge.
- You should always do the performance tuning with DB experts.
- You should always do the performance tuning

Architecture

SessionFactory

- Maintains services that Hibernate uses for all Sessions such as second level caches, connection pools, transaction system integrations etc....
- Thread safe (immutable).
- Expensive to create.
- One per databases.
- https://docs.jboss.org/hibernate/orm/4.2/javadocs/org/hibernate/SessionFactory.html

Session

- Single threaded
- Short lived
- "A Unit of Work"
- Repeatable read persistence context (first level cache).
- Cheap to create.
- https://docs.jboss.org/hibernate/orm/4.2/javadocs/org/hibernate/Session.html

Connection pooling

- Do not use Hibernate's own connection pooling.
- Use an enterprise product.
- These products vary from year to year.

Hibernate's internal connection pooling

- Do not use this for production
- Do not use hibernate.connection.pool_size property, replace it with a third party pool property.

c3p0 connection pool

- org.hibernate.service.jdbc.connections.internal.C3P0ConnectionProvider
- Use the hibernate.c3p0.* properties

Proxool connection pool

- org.hibernate.service.jdbc.connections.internal.ProxoolConnectionProvider
- Use the hibernate.proxool.* properties

Use an application server managed connection pool

- Preferred
- hibernate.connection.datasource

Dialects

- Sometime you have to extend the dialect to use DB specific elements.
- E.g. hibernate in memory pagination.

```
public class MySQLServerDialect extends SQLServerDialect
{
 public MySQLServerDialect()
 {
  registerFunction( "firstrow", new SQLFunctionTemplate( null, " top 1 ?1 " )
  {
   @Override
   public Type getReturnType( Type columnType, Mapping mapping ) throws QueryException
   {
    return columnType;
   }
  } );
 }
}
```

- DDL is vendor specific and comes with the dialect.
- Always check your dialects default type mappings and know your best practices of your DB.
- You can create your own dialect to define your defaults:

```
public class MySQLServerDialect extends SQLServerDialect
{
 public MySQLServerDialect()
 {
   registerColumnType( Types.LONGNVARCHAR, "nvarchar(MAX)" );
 }
}
```

Custom User Types

- We can gain performance by using custom user types.
- Keep in mind the followings:
  - UserType#isMutable() should return false.
  - UserType#nullSafeGet() should return the instance from memory. E.g. from a static, immutable collection.

```
public Object nullSafeGet(ResultSet rs, String[] names, Object owner) throws
HibernateException, SQLException
{
 Integer id = new Integer(rs.getInt(names[0]));
 return rs.wasNull() ? null : MyUserTypes.getInstance(id); // Here we get the instance
from an
}
```

  - UserType#nullSafeSet() should use the id of the entity.

```
public void nullSafeSet(PreparedStatement pst, Object value, int index) throws
HibernateException, SQLException
{
 if (value == null) {
  pst.setNull(index, Types.INTEGER);
 } else {
  pst.setInt(index, ((MyUserType)value).getId().intValue() );
 }
}
```

Read-only entities

- In case of read-only entities Hibernate will NOT
  - dirty-check the entity's simple properties or single-ended associations;
  - update simple properties or updatable single-ended associations.
  - update the version of the read-only entity if only simple properties or single-ended updatable associations are changed.
- In case of read-only entities Hibernate will
  - cascades operations to associations as defined in the entity mapping.
  - updates the version if the entity has a collection with changes that dirties the entity.
- A read-only entity can be deleted.
- Hibernate saves
  - execution time by not dirty-checking simple properties or single-ended associations.
  - saves memory by deleting database snapshots.

Entities of immutable classes

- Instances of immuatable classes will be read only when they are persisted.
- They can be created and deleted.
- Entities of immutable classes are automatically loaded as read-only.

Loading instances of mutable classes as read-only

- Session#setDefaultReadOnly(true)
- If Session#isDefaultReadOnly() returns true the following methods will return read-only entities:
- Session#load()
- Session#get()
- Session#merge()

Loading read-only entities from an HQL query/criteria

- Entities of immutable classes are automatically loaded read-only.
- Use the Query#setReadOnly(true) to return read-only entities.
- Use the Query#setReadOnly(true) before the Query#list(), Query#uniqueResult(), Query#scroll(), Query#iterate()
- Use the Criteria#setReadOnly(true) to return read-only entities.
- Uset the Criteria#setReadOnly(true) before the Criteria#list(), Criteria#uniqueResult(), Criteria#scroll()
- Entities already in the Session are not effected.

Conceptual Session

- SessionFactory.getCurrentSession()
- This is now pluggable.
- Three built in implementations:
  - JTASessionContext
    - https://docs.jboss.org/hibernate/orm/4.2/javadocs/org/hibernate/context/internal/JTASessionContext.html
    - Scopes the notion of the current session to a JTA transaction
    - This implementation will generate Sessions as needed provided a JTA transaction is in effect
    - Returned sessions are automatically configured with both the auto-flush and auto-close attributes set to true, meaning that the Session will be automatically flushed and closed as part of the lifecycle for the JTA transaction to which it is associated.
    - Returned sessions configured to aggressively release JDBC connections after each statement is executed.
    - Short name: jta
  - ThreadLocalSessionContext:
    - https://docs.jboss.org/hibernate/orm/4.2/javadocs/org/hibernate/context/internal/ThreadLocalSessionContext.html
    - The current sessions are tracked by thread of execution.
    - It generates a session upon first requests and cleans it up after the associated Transaction is commited/roll-backed.
    - It is the default.
    - Short name: thread
  - ManagedSessionContext:
    - https://docs.jboss.org/hibernate/orm/4.2/javadocs/org/hibernate/context/internal/ManagedSessionContext.html
    - Current sessions are tracked by thread of execution. However, you are responsible to bind and unbind a Session instance with static methods on this class: it does not open, flush, or close a Session.
    - Short name: managed

Persistence Contexts

- Every Session is associated with a Persistence Context.
- The Persistence Context keeps track of the associated entities.
- The Persistence Context acts as a cache.

When the SQLs are issued?

- You need to know when the SQLs are fired.
- The lifecycle of a Session is bounded by by the beginning and end of a logical transaction.
- Changes to the Persistence Context are detected at Session flush time.
  - Session#flush()
  - Transaction#commit()
- The changes of the Session#flush() will only be seen by other transactions when the transaction attached to the Session is commited.
- It can become relevant when you want to do batch processing.

Customizing the mapping files

- Use the mapping elements and attributes provided by hibernate for customizing the mapping files.
- This will give performance optimized schema.
- The most important elements and attributes:
  - sql-type
  - not-null

- unique
- unique-key
- index
- foreign-key

Lazy Loading

- Lazy loading is a design pattern where we defer the initialization of an object until the point when it is needed.
- Usually it is implemented by proxies.
- Lazy loading plays a crucial role in performance optimization.
- For loading lazy properties you need an attached

Get entities w/o initializing their the data

- For relationships we only need the keys of the objects in the relationship.
- Use the IdentifierLoadAccess#getReference(id)
  - https://docs.jboss.org/hibernate/orm/4.2/javadocs/org/hibernate/IdentifierLoadAccess.html#getReference%28java.io.Serializable%29
  - It returns with a non-nitialized persistence instance with the given identifier.
  - The identifier should exist.
  - It prefers to create a proxy instead of hitting the DB if it is not in the session-> So it does not check the existence.
  - Use it to create foreign-key based associations.
- Example:

```
session.byId( Author.class ).getReference( authorId );
```

Get entities with their data

- IdentifierLoadAccess#load(id)
- https://docs.jboss.org/hibernate/orm/4.2/javadocs/org/hibernate/IdentifierLoadAccess.html#load%28java.io.Serializable%29
- It returns with the fully initialized persistence instance with the given identifier
- Example:

```
session.byId( Author.class ).load( authorId );
```

Native Queries

- When you let Hibernate to create the queries it will know what query will effect which cache.
- In case of Native queries Hibernate will invalidate the second level cache because it will not know the effect of the query.
- Fortunately Hibernate let you specify entities or query spaces that are effected by a natvie query.
- Use the SQLQuery#addSynchronizedEntityName(String)
  - to add a query space (table name) for (a) auto-flush checking and (b) query result cache invalidation checking
  - https://docs.jboss.org/hibernate/orm/4.2/javadocs/org/hibernate/SQLQuery.html#addSynchronizedEntityName(java.lang.String)
- Use the SQLQuery#addSynchronizedQuerySpace(String) to add an entity name for (a) auto-flush checking and (b) query result cache invalidation checking.
  - https://docs.jboss.org/hibernate/orm/4.2/javadocs/org/hibernate/SQLQuery.html#addSynchronizedQuerySpace(java.lang.String)
- User the SQLQuery#addSynchronizedEntityClass(Class) to add an entity for (a) auto-flush checking and (b) query result cache invalidation checking.
  - https://docs.jboss.org/hibernate/orm/4.2/javadocs/org/hibernate/SQLQuery.html#addSynchronizedEntityClass(

java.lang.Class)

Example:

```
SQLQuery sqlQuery = session.createSQLQuery("UPDATE CUSTOMER SET ... WHERE
...");sqlQuery.addSynchronizedEntityClass(Person.class);
int updatedEntities = sqlQuery.executeUpdate();
```

Report Queries

- Sometimes we do not need whole entities. (E.g. for reports).
- We can ignore transactional overhead and dirty checking. (Small gain)
- The idea is that we only query the necessary data.
- It utilizes DB grouping and aggregation.
- It gives you a way to avoid Session cache.
- It is really useful when you have a top level entity (aggregate root) with many bi-directional relation. -> Only some attributes will be fetched instead of very complex objects.

```
// Naive approach
from Company company
 left join fetch company.branch
 left join fetch company.branch.city
    ....

// Clever approach
select branch.company.name, branch.name, branch.city.name
from Branch branch
```

HQL syntax:

```
[select ...] from ... [where ...]
[group by ... [having ...]] [order by ...]
```

The dynamic query returns an Object[] for each row:

```
Iterator i = session.createQuery(
  "select stock.id, stock.shortName, tr.amount " +
  "from Stock stock join stock.transactions tr " +
  "where tr.amount > 100" ).list().iterator();

while ( i.hasNext() )
{
 Object[] row = (Object[]) i.next();
 Long id = (Long) row[0];
 String shortName = (String) row[1];
 BigDecimal amount = (BigDecimal) row[2];
 // ...
}
```

We can return new entities:

```
Iterator i = session.createQuery(
  "select new StockTransaction( stock.id, stock.shortName, tr.amount ) " + // StockTransaction should
not be a persistent class.
  "from Stock stock join stock.transactions tr " +
  "where tr.amount > 100" ).list().iterator();

while ( i.hasNext() )
{
 StockTransaction row = (StockTransaction) i.next();
 // ...
}
```

Aggreate functions:

- The semantic is the same as their SQL counterpart. The supported aggregate functions are:
  - `COUNT` (including distinct/all qualifiers) - The result type is always Long.
  - `AVG` - The result type is always Double.
  - `MIN` - The result type is the same as the argument type.
  - `MAX` - The result type is the same as the argument type.
  - `SUM` - The result type of the `avg()` function depends on the type of the values being averaged. For integral values (other than BigInteger), the result type is Long.
    For floating point values (other than BigDecimal) the result type is Double. For BigInteger values, the result type is BigInteger. For BigDecimal values, the result type is BigDecimal.

```
select min(tr.amount), max(tr.amount)
from Transaction tr where tr.stock.id = 1
```

Grouping

- Use the group by and having keywords

```
select tr.stock.id, sum( tr.amount)
from Transaction tr
group by tr.stock.id
```

Transactions

- Transaction flushes the Session.
- The Session object's persistence context will contain a reference to all entities associated with the transaction. -> It can lead to memory problems.
- Hibernate uses JDBC
- Hibernate does not lock objects in memory.
- The isolation level of your database does not change when you use Hibernate.
- The org.hibernate.Session acts as a
  - transaction-scoped cache providing repeatable reads by identifiers
  - queries that result in loading entities.

Transactional write-behind

- The physical database transaction needs to be as short as possible.
- Long database transactions prevent your application from scaling to a highly-concurrent load.
- Do not hold a database transaction open during end-user-level work, but open it after the end-user-level work is finished.
- This is concept is referred to as transactional write-behind.

Transactional patterns

Session-per-operation antipattern

- It means openeing and closing a session for each DB operation in a single thread.
- Hibernate automatically disables auto-commit.

Session-per-request pattern

- Hibernate opens the session for the request -> Starts the transaction -> Performs all tasks -> Ends the transaction.
- One-to-one relationship between the transaction and the session.
- This pattern defines the current session.
- SessionFactory#getCurrentSession()
  - https://docs.jboss.org/hibernate/orm/4.2/javadocs/org/hibernate/SessionFactory.html#getCurrentSession%28%29

Conversational pattern

- We have a conversational scope that spans multiple requests (E.g. Stateful Session Beans).
- Do not start a transaction at the start and keep it open for the duration of the conversation. -> Lock contention.
- Maintaining the isolation is part of the business tier.
- Hibernate support:
  - Automatic optimistic concurrency control with automatic versioning.
  - Detached objects.
  - The Hibernate Session can be disconnected from the underlyin JDBC connection and can be recconnected.

Session-per-application and Session-per-user-session antipatterns

- Two object instances representing the same entity can be inconsistent.
- High memory requirement.
- The Session is not thread-safe.
- A Hibernate exception will close the session. You should close the applicaiton/session.

Batch Processing

One-by-one processing

- Hibernate operates on objects one-by-one.
- It requires a large number of database roundtrips.
- It is bad in one-to-many associations.
- There are some SQL operations that can work on multiple records. <-> In hibernate this is accomplised on a per entity bases.
- E.g. One SQL Update on a table is one Update on every entity.
- It is really bad for one-to-many, manay-to-many associations.
- It can result in out of memory. The session contains every attached object.

DML-syle operations:

- HQL supports DML-style insert, delete and update.

UPDATE, DELETE

- Pseudo code:
  - (UPDATE | DELETE) FROM? EntityName (WHERE where_condition)?
  - The ? suffix indications an optional parameter. The FROM and WHERE clauses are each optional.
- Example for update:

```
update Zoo set name = :newName where name = :oldName
```

- Example for delete:

```
delete Zoo z where z.name = :name
```

- The update by default does not effect the version or the timestamp property values for the effected entities.
- Use the versioned keyworkd after the UPDATE keyworkd to reset the version or timestamp properties.

```
update versioned Zoo set name = :newName where name = :oldName
```

- The delete can result more delete SQL statements based on the DB representation of the entity (e.g. it is stored in a more tables).

INSERT

- Pseudo code:
  - INSERT INTO EntityName properties_list select_statement
- Example:

```
insert into DelinquentAccount (id, name) select c.id, c.name from Customer c where ...
```

- The properties_list is analogous with the column specification of the SQL INSERT.
- The_select_statment can be any valid HQL select query, but the return entity type must match with the EntityName in the Insert query.
- If the id is not in the properties_list, hibernate wil generate ids for the new entities.
- The automatic id generation only works with ID generators which operate on the database.

JDBC batching

- Batch Processing allows you to group related SQL statements into a batch and submit them with one call to the database.
- Set the property hibernate.jdbc.batch_size to an integer between 10 and 50. This will tell hibernate that every X rows are to
  be inserted as batch.
- JDBC batch can target only one table. -> Every new DML statement with a different talbe ends up the current batch and starts a new one. ->
  Do not mix different tables.
- Sort the inserts and updates with the following Hibernate properties:
  - hibernate.order_inserts
    - Values: true|false
  - hibernate.order_updates
    - Values: true|false
    - Forces Hibernate to order SQL updates by the primary key value of the items being updated. This reduces the likelihood of transaction deadlocks in highly-concurrent systems.
- To add update batching set the hibernate.jdbc.batch_versioned_data to true.
  - Set this property to true if your JDBC driver returns correct row counts from executeBatch(). This option is usually safe, but is disabled by default. If enabled, Hibernate uses batched DML for automatically versioned data.
  - Check your driver!!!
- Batch processing of large number of objects can cause memory problems because of the first-level cache.
- Use Session#flush() and Session#clear() regularly to avoid memory problems.
- JDBC batch opetations of java.sql.Statement, java.sql.PreparedStatement, java.sql.CallableStatement:
  - #addBatch()
  - #executeBatch()
  - #clearBatch()
- Set auto-commit false.
- Hibernate disables insert batching at the JDBC level transparently if you use an identity identifier generator.
- If the above approach is not appropriate, you can disable the second-level cache, by setting hibernate.cache.use_second_level_cache to false.

Batch inserts

- Use the Session#flush() and Session#clear() to control the size of the first-level cache.
- Example:

```
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();
```

Batch updates

- Use the Session#flush() and Session#clear() to control the size of the first-level cache.
- Use the ScrollableResult#scroll() to take advantage of the server-side cursor.
- Example:

```
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();
```

Batch deletes

- By default deletes are not batched.
- Use SQL cascade deletion.
- Mark your one-to-many association with @OnDelete
- OnDelete
  - https://docs.jboss.org/hibernate/orm/4.2/javadocs/org/hibernate/annotations/OnDelete.html
  - Strategy to use on collections, arrays and on joined subclasses delete
  - OnDelete of secondary tables currently not supported.
- OnDeleteAction
  - https://docs.jboss.org/hibernate/orm/4.2/javadocs/org/hibernate/annotations/OnDeleteAction.html
  - Possible actions on deletes.
    - OnDeleteAction#CASCADE
    - OnDeleteAction#NO_ACTION

Example:

```
// One-to-many
@OneToMany(cascade = {
      CascadeType.PERSIST,
      CascadeType.MERGE}, // We do not want Hibernate to propagate the entity removal. We will do
this with @OnDelete.
   mappedBy = "user")
@OnDelete(action = OnDeleteAction.CASCADE)
private List<Email> emails = new ArrayList<>();

// Many-to-one@OneToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "id")
@MapsId
@OnDelete(action = OnDeleteAction.CASCADE)
private Post post;
```

Example2

```
public class House {

    @OneToOne
    Object door;
}
```

If you use CascadeType.REMOVE then deleting the house will also delete the door (using an extra SQL statement).

```
    @OneToOne(cascade=CascadeType.REMOVE)
    Object door;
```

If you use @OnDelete then deleting the door will also delete the house (using an ON DELETE CASCADE database foreign key).

```
    @OneToOne
    @OnDelete(action = OnDeleteAction.CASCADE)
    Object door;
```

StatelessSession

- Statelessness is the key for scaling multi-user applications.
- Hibernate provides the StatelessSession for this.
- https://docs.jboss.org/hibernate/orm/4.2/javadocs/org/hibernate/StatelessSession.html
- A command-oriented API for performing bulk operations against a database.
- It does not have a first-level cache.
- It does not interact with the second-level cache and with the query cache.
- It does not implement transactional write-behind.
- It does not do automatic dirty checking.
- Collections are ignored. -> Do not manipulate either the parent or the child objects.
- Operations performed via a stateless session bypass the event model and interceptors.
- Operations are never cascaded.
- For certain operations a stateless session may be faster than a stateful session.
- Example:

```
StatelessSession session = sessionFactory.openStatelessSession();
try{
// do all operations here
}
...
...
finally{
        session.close();
}
```

```
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); // he Customer instances returned by
the query are immediately detached. They are never
        associated with any persistence context.
    customer.updateStuff(...);
    session.update(customer);
}

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

Locking

- 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.
- Your locking strategy can be either optimistic or pessimistic.
- Pessimistic locking:
    - Most of the times multiple transactions can complete with each other.
    - The resources is locked when the transaction locked. Other transactions cannot access the resource.
    - The lock exists until the transaction is finished.
- Optimistic locking:
    - Most of the times multiple transaction can complete w/o effecting each other.
- NOTE: transaction isolation level

Optimistic locking

- Most of the times multiple transaction can complete w/o effecting each other.
- The resource is not actually locked.
- Other transactions can access the resource and conflicting changes are possible.
- At commit time, when the resource is about to be updated in persistent storage, the state of the resource is read from storage again and compared to the state that was saved when the resource was first accessed in the transaction. If the two states differ, a conflicting update was made, and the transaction will be rolled back.
- This is good for read heavy applications where Read >>> Write.
- We can use version number or timestamp to keep track of the changes.
- Hibernate treats null version numbers / timestamps as transient objects.

Version number

- Use the java.persistence.Version annotation to specifiy the version field or property of an entity class that serves as its optimistic lock value
- https://docs.oracle.com/javaee/7/api/javax/persistence/Version.html
- Only one version property / class.
- The version property should be mapped to the primary table
- The following types are supported for version properties: int, Integer, short, Short, long, Long, java.sql.Timestamp. NOTE: timestamping is less reliable.
- The application must not change the version number.
- If the version number is generated by the database, such as a trigger, use the annotation @org.hibernate.annotations.Generated(GenerationTime.ALWAYS).

```
@Entity
public class Account implements Serializable
{
 ...

 @Version
 public Long getVersion(){ ... } ;

 ...
}
```

Pessimistic Locking

- Specify the isolation level for the JDBC connection and let the DB handle the locking.
- Hibernate always uses the locking mechanism of the DB and never locks objects in memory.
- Use the following actions to lock resources:
  - Session#load(), specifying lock mode
  - Session#lock()
  - Query#setLockMode()
- Lock modes:
  - LockMode.WRITE
    - Acquired for update and insert
  - LockMode.UPGRADE
    - Acquired on explicit user request using SELECT ... FOR UPDATE SQL syntax.
  - LockMode.UPGRADE_NOWAIT
    - Acquired on explicit user request using SELECT ... FOR UPDATE SQL syntax in Oracle.
  - LockMode.READ
    - Acquired automatically on reads with Repeatable Read or Serializable isolation level.
  - LockMode.NONE
    - The absence of locking.
- If the requested lock mode is not supported by the database, Hibernate uses an appropriate alternate mode instead of throwing an exception. This ensures that applications are portable.

Caching

- First level cache
  - Session cache.
  - Cache objects within the current session.
- Second level cache
  - Caches objects on SessionFactory level (across sessions).
- Query cache
  - Caches queries and their result (IDs)

Cache Mode

- Enum that controls how the session interacts with the second level cache and the query cache.
- https://docs.jboss.org/hibernate/orm/4.2/javadocs/org/hibernate/CacheMode.html
- GET
  - The session may read items from the cache, but will not add items, except to invalidate items when updates occur.
- IGNORE
  - The session will neverinteract with the cache, except to invalidate cache items when updates occur.
- NORMAL
  - The session may read items from the cache, and add items to the cache.
- PUT
  - The session will never read items from the cache, but will add items to the cache as it reads them from the

database.
- REFRESH
  - The session will never read items from the cache, but will add items to the cache as it reads them from the database.
  - In this mode, the effect of hibernate.cache.use_minimal_puts is bypassed, in order to force a cache refresh

Cache Properties

Property	Example	Purpose
hibernate.cache.provider_class	Fully-qualified classname	The classname of a custom CacheProvider.
hibernate.cache.use_minimal_puts	`true` or `false`	Optimizes second-level cache operation to minimize writes, at the cost of more frequent reads. This is most useful for clustered caches and is enabled by default for clustered cache implementations.
hibernate.cache.use_query_cache	`true` or `false`	Enables the query cache. You still need to set individual queries to be cachable.
hibernate.cache.use_second_level_cache	`true` or `false`	Completely disable the second level cache, which is enabled by default for classes which specify a <cache> mapping.
hibernate.cache.query_cache_factory	Fully-qualified classname	A custom `QueryCache` interface. The default is the built-in `StandardQueryCache`.
hibernate.cache.region_prefix	A string	A prefix for second-level cache region names.
hibernate.cache.use_structured_entries	`true` or `false`	Forces Hibernate to store data in the second-level cache in a more human-readable format.
hibernate.cache.use_reference_entries	`true` or `false`	Optimizes second-level cache operation to store immutable entities (aka "reference") which do not have associations into cache directly, this case, lots of disasseble and deep copy operations can be avoid. Default value of this property is `false`.

Session (first-level) cache

- Caches values within the current session.
- There can be a lot of session objects
  - -> They should be short lived
  - -> They should store only a few objects.
- Session#get(), Session#load() interact with the first-level cache.
- Queries do not interact with the first-level cache.
- Use the Session#evict() to remove an entity from the cache.
  - https://docs.jboss.org/hibernate/orm/4.2/javadocs/org/hibernate/Session.html#evict%28java.lang.Object%29

Interacting with the cache

- Saving and updating
  - Session#save()
  - Session#update()
  - Session#saveOrUpdate()
- Retrieving
  - Session#load()
  - Session#get()
  - Session#list()
  - Session#iterate()
  - Session#scroll()
- You can remove an item from the cache with the Session#evict(Object)
- The Session#clear() removes all item from the cache.

Session#get() vs Session#load()

- Session#get
  - https://docs.jboss.org/hibernate/orm/4.2/javadocs/org/hibernate/Session.html#get%28java.lang.Class,%20java.io.Serializable%29
  - It does not assumes existence.
  - Return the persistent instance of the given entity class with the given identifier, or null if there is no such persistent instance.
  - If the instance is already associated with the session, return that instance. This method never returns an uninitialized instance.
- Session#load()
  - https://docs.jboss.org/hibernate/orm/4.2/javadocs/org/hibernate/Session.html#load%28java.lang.Class,%20java.io.Serializable%29
  - Returns with the persistent instance.
  - Assumes existence -> No SQL Select -> It returns with a proxy
  - Do not use this for existence checking.

Second level cache

- Hibernate is compatible with several cache providers.
- The second level cache belongs to the SessionFactory.
- You can configure caching on a class-by-class and collection-by-collection basis

Enable caching

- By default entities are not cached.
- Cache those items that
  - do not change often.
  - often needed.
  - have only a few instances (Some thousands can be Ok. Millions Not)
- Use the @Cacheable on the entity

@Cacheable

- https://docs.oracle.com/javaee/7/api/javax/persistence/Cacheable.html
- Specifies whether an entity should be cached if caching is enabled when the value of the `persistence.xml` caching element is `ENABLE_SELECTIVE` or `DISABLE_SELECTIVE`.
- `The value of the Cacheable annotation is inherited by subclasses.`
- `It can be overridden by specifying Cacheable on a subclass.`
- `Example:`

```
@Entity
@Cacheable
@Cache(usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
public class User { ... }
```

`@Cache`

- `Complement the @Cacheable.`
- `It specifies how to interact with the cache.`
- `Add caching strategy to a root entity or a collection`
- https://docs.jboss.org/hibernate/orm/4.2/javadocs/org/hibernate/annotations/Cache.html
- CacheConcurrencyStrategy
  - https://docs.jboss.org/hibernate/orm/4.2/javadocs/org/hibernate/annotations/CacheConcurrencyStrategy.html
  - usage
    - NONE
    - READ_ONLY
      - The region will be read-only.
      - No locks are needed -> performant.
      - When the entity persisted only the database contains a copy of this entity
      - The cache the entity when it is first fetched.
      - All subsequent call will get the entity from the cache.
      - If you modify any of the objects will get an excption.

        - Entities are evicted from the cache upon deletion.
    - NONSTRICT_READ_WRITE
        - The cacheable objects can be modified but the cache provider doesn't need to implement a lock on the objects.
        - It can result in stale data.
    - READ_WRITE
        - The cacheable object can be modified and the cache provider will use a lock when the object is updated.
        - No stale objects.
        - It is a good choice for write intensive applications.
        - It is asynchronous.
        - INSERT
            - The newly created entities will be put into the cache right after the db transaction commit. (After insert with a write lock on the region of the entity)
            - There can be a cache synchronization lag. The lag is between the DB commit and Java cache insert.
              During this time another process can modify the entity.
        - UPDATE and DELETE
            - More complex than the INSERT
            - A lock object is used to prevent reading the cache entry.
            - Heavy write contention the locking construct will allow other concurrent transactions to hit the database (in case of read operations).
        - Controlling the timeout
            - `It can be contolled with custom cache properties (see the vendor's documentation).`
            - `net.sf.ehcache.hibernate.cache_lock_timeout`
    - TRANSACTIONAL
        - Valid for distributed caches.
        - It is synchronized with the underlying XA transaction.
        - So it ensures integrity with synchronously.
        - Less performant than READ_WRITE.
- region:
    - Name of the cache region.
    - By default it is the fully qualified name of the class or the fully qualified role name of the collection.
- include:
    - all: includes all properties. Default
    - non-lazy: only non-lazy attributes.

Example with Locking and Second level cache!!!!

Ineractions between first and second level cache

- The CacheMode controls the interaction between the first and second level cache.
- https://docs.jboss.org/hibernate/orm/4.2/javadocs/org/hibernate/CacheMode.html
- CacheMode.NORMAL
    - reads items from and writes them to the second-level cache.
- CacheMode.GET
    - reads items from the second-level cache, but does not write to the second-level cache except to update data.
- CacheMode.PUT
    - writes items to the second-level cache. It does not read from the second-level cache. It bypasses the effect of hibernate.cache.use_minimal_puts and forces a refresh of the second-level cache for all items read from the database.

See the statistics of the second level cache

- Set hibernate.generate_statistics to true.
- Set hibernate.cache.use_structured_entries to true to store cache entries in a human readable format.
- Example:

```
Map cacheEntries = sessionFactory.getStatistics()
        .getSecondLevelCacheStatistics(regionName)
        .getEntries();
```

Query cache

- We cache the queries with their parameters and their results (ID's of the result entities).
- It can be good or bad for you app.
- It should always be evaluated in the context of the current application.
- Enable query cache on a query bases.
    - Evaluate all queries.
    - Do not put all queries into the query cache.
- Query cache does NOT cache entity states, only identifiers. -> Use it with the second-level cache.

Query cache can consume a lot of memory.

- Query cache contains queries with their parameters.
- QueryKey
    - https://docs.jboss.org/hibernate/orm/4.2/javadocs/org/hibernate/cache/spi/QueryKey.html
    - Identifies a query with bound parameter values.
    - It is used as a key in the query cache.
    - It can cause a problem if the parameters are entities. it will hold a reference to these entities. -> No garbage collection.
- Example:

```
// Problem
final Cat mate = ...;
final String hql = "from Cat as cat where cat.mate = ?"
final Query q = session.createQuery(hql);
q.setParameter(0, mate); // The QueryKey will hold a reference to the mate :(
q.setCacheable(true);


// Solution.
// Do not use whole entities as parameters
final Cat mate = <...>;
final String hql = "from Cat as cat where cat.mate.id = ?"
final Query q = session.createQuery(hql);
q.setParameter(0, mate.getId());
q.setCacheable(true);
```

- Example:

```
// Problem
// We use entities in our query as parameters
final String foo = "something";
final Cat mate = ...;

final Criteria crit;
final NaturalIdentifier natId;

crit = session.createCriteria(Cat.class);
natId = Restrictions.naturalId();
natId.set("mate", mate);
natId.set("foo", foo);
crit.add(natId);
crit.setCacheable(true);// Solution// Use only properties instead of whole objectsfinal String foo
= "something";
final Cat mate = ...;

final Criteria crit;
final NaturalIdentifier natId;

crit = session.createCriteria(Cat.class);
natId = Restrictions.naturalId();
natId.set("mate.id", mate.getId());
natId.set("foo", foo);
crit.add(natId);
crit.setCacheable(true);
```

- Query cache doesn't use hashCode() and equals()
  - You can write your own implementation if you want.
  - Decorate the default StandardQueryCache and override the StandardQueryCache#put().
  - https://docs.jboss.org/hibernate/orm/4.2/javadocs/org/hibernate/cache/internal/StandardQueryCache.html

Enabling the Query Cache

- Set the hibernate.cache.user_query_cache property to true.
- This enables two cache regions:
  - org.hibernate.cache.internal.StandardQueryCache:
    - holds the cached query results.
  - org.hibernate.cache.spi.UpdateTimestampsCache:
    - holds timestamps of the most recent updates to queryable tables.
    - These timestamps validate results served from the query cache. It should be higher expiry or timeout than the org.hibernate.cache.internal.StandardQueryCache region.
    - Do not use LRU for the cache region.
    - If possible set it to never expire.

Caching a query regions

- Set a query cacheable with the Query#setCacheable()
  - https://docs.jboss.org/hibernate/orm/4.2/javadocs/org/hibernate/Query.html#setCacheable%28boolean%29
- Query cache regions can be specified on query level. with Query#setCacheRegion()
  - https://docs.jboss.org/hibernate/orm/4.2/javadocs/org/hibernate/Query.html#setCacheRegion%28java.lang.String%29
- Control the cache mode with the Query#setCacheMode()
  - https://docs.jboss.org/hibernate/orm/4.2/javadocs/org/hibernate/Query.html#setCacheMode%28org.hibernate.CacheMode%29
- To force a query cache refresh to disregard any cached results in the regions
  call org.hibernate.Query.setCacheMode(CacheMode.REFRESH)
  It is more efficient than org.hibernate.SessionFactory.evictQueries()

Primary Key Generation

Choosing Primery key

Simple vs Compund keys

- Simple key (It ususually has an associated default index) -> Faster
- Compund key -> More complex index -> Slower

Numeric vs Non-numeric key

- Numberic is faster

Natural vs Surrogate key

- Surrogate key is more stable.
- Natural key is not stable because of the schema evolution.
- Surrogate keys:
  - Database generated vs Java generated
  - Java generated is good for batch processing (GUID or UUID)

UUID generation

- UUID can be used as surrogate keys.
- It is good because we can generate them from Java.

Database support storage

- Some DB offers a dedicated UUID type
- It can be stored as byte array. E.g. BINARY(16)
- It can be stored as two bigint columns.
- It can be stored as the hex value with CHAR(36)

UUID generation strategies

- UUID Assignment
  - Create an UUID field and assign a new UUID() to it.

```
@Entity
public static class User
{

   @Id
   @Column(columnDefinition = "BINARY(16)")
   private UUID uuid;

   public User() {}

   public User(UUID uuid)
 {
      this.uuid = uuid;
   }
}



session.persist( new User( UUID.randomUUID() ); // 1 insert
session.flush();session.merge( new User( UUID.randomUUID() ); // 1 select + 1
insertsession.flush();
```

  - Session#merge() will issue a select, then an insert. It needs a select to see if there is already an entity in DB with this UUID.
    For other identifier generations Hibernate looks for null identifiers to see if it is transient or not. Some frameworks, e.g Spring Data SimpleJpaRepository#save() will always

use merge in this case so it will always results in one SELECT + one SAVE.

```
@Transactional
public <S extends T> S save(S entity) {
    if (entityInformation.isNew(entity)) {
        em.persist(entity);
        return entity;
    } else {
        return em.merge(entity);
    }
}
```

- UUID generators
  - We let Hibernate to generate the UUID.
  - If null found Hibernate assumes it a transient and it will generate a new identifier. -> Session#merge() does not issue a SELECT.
  - UUIDHexGenerator
    - 32 hexadecimal UUID string.
    - Example:

```
@Entity
public static class User{

    @GeneratedValue(generator = "uuid")
    @GenericGenerator(name = "uuid", strategy = "uuid")
    @Column(columnDefinition = "CHAR(32)")
    @Id
    private String id;
}

session.persist(new User());
session.flush(); // 1 insert
session.merge(new User());
session.flush(); // 1 insert
```

  - UUIDGenerator
    - Supported types:
      - java.lang.UUID
      - byte[16]
      - String (Hexa)
    - Example

```
@Entity
public static class User
{

   @GeneratedValue(generator = "uuid2")
   @GenericGenerator(name = "uuid2", strategy = "uuid2")
   @Column(columnDefinition = "BINARY(16)")
   @Id
   private UUID id;
}

session.persist(new User());
session.flush(); // 1 insert
session.merge(new User());
session.flush(); // 1 insert
```

Autogeneration Strategies

- Some databases allowes you to choose between several strategies.
- The PK generation should be independent from the current transaction
- Hibernate disables JDBC batch inserts in case of IDENTITY generator strategy.

javax.persistence.GenerationType.AUTO

- The DB should pick the strategy.
- Do not use this.

javax.persistence.GenerationType.IDENTITY

- Indicates that the persistence provider must assign primary keys for the entity using a database identity column.

javax.persistence.GenerationType.SEQUENCE

- The persistence provider must assign primary keys for the entity using a database sequence.
- You can use cached DB sequences.
- You can use Java generated sequences. E.g. Hi/Lo algorythm

javax.persistence.GenerationType.TABLE

- The persistence provider must assign primary keys for the entity using an underlying database table to ensure uniqueness.

Identifier optimizers for sequence identifiers

Hi/Lo algorythm

It splits the sequences domain into hi groups. The hi value is obtained synchronously.

Every hi group has a certain number of low entries. Lo entries can be used - in the context of a hi value - off-line.

1. The "hi" token is obtained from DB. This should be thread safe.
2. We need the increment-size for the low entries -> it gives the number of lo entries.
3. Use the low values.
   Low value range (identifiers) [(hi - 1) * incrementSize)+1, (hi * incrementSize) + 1)
   Low values are taken from [0,incrementSize] starting at [(hi-1) * incrementSize) + 1)
4. When all lo valuds are used, we need to obtain a new hi value.

Generators:

- SequenceHiLoGenerator
  - It uses a DB sequence to generate the hi value, the low value is incremented according to the hi/low algorythm
  - https://docs.jboss.org/hibernate/orm/4.2/javadocs/org/hibernate/id/SequenceHiLoGenerator.html
- MultipleHiLoPerTableGenerator
  - A HiLo identifier generator that uses a DB table to store generated hi values.
  - This is not compliant with a user provided connectin.
  - https://docs.jboss.org/hibernate/orm/4.2/javadocs/org/hibernate/id/MultipleHiLoPerTableGenerator.html
- SequenceStyleGenerator
  - It uses a sequence if the underlying DB supports them.
  - Otherwise it uses a DB table for storing sequence values.
  - https://docs.jboss.org/hibernate/orm/4.2/javadocs/org/hibernate/id/enhanced/SequenceStyleGenerator.html
  - It can be configured to use an opmization strategy:
    - none: No optimization. Every id is fetched from DB.
    - hi/lo: it uses the hi/lo algorythm
    - pooled: This optimizer uses a hi/lo optimization strategy, but instead of saving the current hi value it stores the current range upper boundary (or lower boundary –hibernate.id.optimizer.pooled.prefer_lo). this is the default optimizer
- TableGenerator
  - An enhanced table-based id generation.
  - A single table can actually server for the storage of multiple independent generators.
  - It supports multiple optimization strategies. The pooled is the default.
  - https://docs.jboss.org/hibernate/orm/4.2/javadocs/org/hibernate/id/enhanced/TableGenerator.html

Example:

```
@Entity(name = "users")
public static User PooledLoSequenceIdentifier {

   @Id
   @GenericGenerator(name = "sequenceGenerator", strategy = "enhanced-sequence",
         parameters = {
               @org.hibernate.annotations.Parameter(name = "optimizer",
                     value = "pooled-lo"
               ),
               @org.hibernate.annotations.Parameter(name = "initial_value", value = "1"),
               @org.hibernate.annotations.Parameter(name = "increment_size", value = "5")
         }
   )
   @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "sequenceGenerator")
   private Long id;
}
```

Associated objects and fetching strategies

- Fetching strategies are used to fine-tune the fetching the associated objects.
- It determines the number of queries that Hibernate uses to fetch associations from DB.
- We use the fetching strategies to improve performance.
- They can declared in O/R metadata.
- The fetching related O/R metadata can be overridden by on Query of Criteria bases.
- Fetching strategies:
  - Join fetching
  - Select fetching

    - Subselect fetching
    - Batch fetching
- Note: @ManyToOne, @OneToOne is eager by default.
- hibernate.max_fetch_depth
    - Sets a maximum "depth" for the outer join fetch tree for single-ended associations (one-to-one, many-to-one). A 0 disables default outer join fetching. e.g. recommended values between 0 and 3.
    - This configuration property determines how many associations hibernate will traverse by join when fetching data
    - This can be bad or good. There is always a point when a seaparate select would be better.
- hibernate.default_batch_fetch_size
    - Sets a default size for Hibernate batch fetching of associations. e.g. recommended values 4, 8, 16
    - It is used by Hibernate when interacting with the DB via the JDBC batch API.
    - Usually we do not touch this.

Defining fetching strategies

- Via XML metadata
    - Use the fetch attribute of the collection mapping.
    - Example:

```
<set name="addresses"
        fetch="join">
  <key column="personId"/>
  <one-to-many class="Address"/>
</set
```

- Via Java annotations
    - Use the @Fetch annotaion and pass the FetchMode.
    - @Fetch
        - Define the fetching strategy used for the given association
        - Pass the desired FetchMode instance to this annotation.
    - FetchMode
        - Represents an association fetching strategy.
        - This is used together with the Criteria API to specify runtime fetching strategies.
        - For HQL queries, use the FETCH keyword instead.
        - https://docs.jboss.org/hibernate/orm/4.2/javadocs/org/hibernate/FetchMode.html
    - Example

```
public class Person implements Serializable
{
 ...
 @OneToMany(fetch = FetchType.LAZY, mappedBy = "person")
 @Cascade(CascadeType.ALL)
 @Fetch(FetchMode.SELECT)
 public Set<Addresses> getAddresses()
 {
  return this.addresses;
 }
 ...
}
```

- Via HQL
    - Use JOIN FETCH to overide outer join and lazy declarations of the mapping file for associations and collections.
    - Example

```
// Collection mapping
public class Person implements Serializable
{
 @OneToMany( fetch = FetchType.Lazy )
 private Set<Address> addresses;
 ...
}

// Query to eagerly fetch the collection
FROM Person person
JOIN FETCH person.addresses
```

- Criteria#setFetchMode
  - Specify an association fetching strategy for an association or a collection of values.
  - https://docs.jboss.org/hibernate/orm/4.2/javadocs/org/hibernate/Criteria.html#setFetchMode(java.lang.String, org.hibernate.FetchMode)
  - Example:

```
User user = (User) session.createCriteria(Person.class)
            .setFetchMode("addresses", FetchMode.JOIN)
            .add( Restrictions.idEq(personId) )
            .uniqueResult();
```

Lazy vs eager associations

- By default Hibernate uses lazy associations.
- Do not change this. -> You can fetch the whole DB (for every user)
- Lazy associations will throw an exception outside the context of an open Hibernate session. <-> Minimize the allocation of the exernal resources (Close the session asap).
- Best practices:
  - Use lazy associations
  - Close the session asap (in the business tier)
  - Create requirement specific queries that fetch the needed data at once.
  - Do not use open-session in view. It is an anti-pattern.

Join fetching strategy

- fetch=”join” or @Fetch(FetchMode.JOIN)
- It always loads the entities and it's associated objects.
- It is eager.
- Hibernate will use one SQL SELET. It retrieves the entity and all of its related collections with a LEFT OUTER JOIN

```
SELECT ...
FROM entity e
LEFT OUTER JOIN coll1 c1 on e.id = c1.eId
LEFT OUTER JOIN coll2 c2 on e.id = c2.eId
```

- Example:

```
public class Person implements Serializable
{
 ...
 @Fetch(FetchMode.JOIN)
 @OneToMany(cascade=CascadeType.ALL,mappedBy="person",fetch=FetchType.LAZY)
 private Set<Email> emails = new HashSet<Email>();
 ...
}
```

```
System.out.println( "-- SELECT person --");
Person person = (Person)session.find(Person.class, p.getId() );
System.out.println( person );

System.out.println( "-- Iterate emails: --");
for ( Email email: person.getEmails()  )
{
 System.out.println( email );
}
```

```
-- SELECT person --
Hibernate:
    select
        person0_.id as id1_21_0_,
        emails1_.person_id as person_i2_21_1_,
        emails1_.id as id1_7_1_,
        emails1_.id as id1_7_2_,
        emails1_.person_id as person_i2_7_2_
    from
        Person person0_
    left outer join
        Email emails1_
            on person0_.id=emails1_.person_id
    where
        person0_.id=?
com.p92.javacourse.jee.jpa.associations.Person@689b7a6a
-- Iterate emails: --
com.p92.javacourse.jee.jpa.associations.Email@615db116
com.p92.javacourse.jee.jpa.associations.Email@342b76a8
```

Select fetching strategy

- fetch=”select” or @Fetch(FetchMode.SELECT)
- It loads only the entity first.
- It loads the associations when they are requested by issuing a new SQL SELECT.
- It is lazy.
- Example:

```
public class Person implements Serializable
{
 ...
 @Fetch(FetchMode.SELECT)
 @OneToMany(cascade=CascadeType.ALL,mappedBy="person",fetch=FetchType.LAZY)
 private Set<Email> emails = new HashSet<Email>();
 ...
}
```

```
System.out.println( "-- SELECT person --");
Person person = (Person)session.find(Person.class, p.getId() );
System.out.println( person );

System.out.println( "-- Iterate emails: --");
for ( Email email: person.getEmails()  )
{
 System.out.println( email );
}
```

```
-- SELECT person --
Hibernate:
    select
        person0_.id as id1_21_0_
    from
        Person person0_
    where
        person0_.id=?
com.p92.javacourse.jee.jpa.associations.Person@104468e2
-- Iterate emails: --
Hibernate:
    select
        emails0_.person_id as person_i2_21_0_,
        emails0_.id as id1_7_0_,
        emails0_.id as id1_7_1_,
        emails0_.person_id as person_i2_7_1_
    from
        Email emails0_
    where
        emails0_.person_id=?
com.p92.javacourse.jee.jpa.associations.Email@285f4e6b
com.p92.javacourse.jee.jpa.associations.Email@59bd80fc
```

Batch select fetching strategy

- batch-size=”10” or @BatchSize(size = 10)
- Batch implementation of the Select fetching strategy.
- It loads N entities or collections instead of one.
- It issues less SQL SELECT than the Select fetching strategy.
- Strategies for determining the number of selects. The strategy can be set by the hibernate.batch_fetch_style property.
    - Possible values: legacy, padded, dynamic
- Legacy batch fetch style
    - Uses pre-built batch sizes based on org.hibernate.internal.util.collections.ArrayHelper#getBatchSizes. E.g. [25,

12, 10, 9, 8, 7, .., 1]
- E.g. 24 collections -> 3 queries (12, 10, 2)

```
select * from owner where id in (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
select * from owner where id in (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
select * from owner where id in (?, ?)
```

- Padded batch fetch style:
  - Similar to the legacy.
  - It uses the closest number from the pre-built batch sizes based on org.hibernate.internal.util.collections.ArrayHelper#getBatchSizes. but repeat some values as padding.
- Dynamic
  - Dynamically builds the SQL based on the actual numer of available ids.
- Example:

```
public class Person implements Serializable
{
 ...
 @Fetch(FetchMode.SELECT)
 @OneToMany(cascade=CascadeType.ALL,mappedBy="person",fetch=FetchType.LAZY)
 @BatchSize(size=3)
 private Set<Email> emails = new HashSet<Email>();
 ...
}
```

```
System.out.println( "-- SELECT person --");
Person person = (Person)session.find(Person.class, p.getId() );
System.out.println( person );

System.out.println( "-- Iterate emails: --");
for ( Email email: person.getEmails()  )
{
 System.out.println( email );
}
```

```
Hibernate:
    /*
FROM
    Person */ select
        person0_.id as id1_21_
    from
        Person person0_
Iterating persons
com.p92.javacourse.jee.jpa.associations.Person@1393abe4
-- Iterate emails: --
Hibernate:
    /* load one-to-many com.p92.javacourse.jee.jpa.associations.Person.emails */ select
        emails0_.person_id as person_i2_21_1_,
        emails0_.id as id1_7_1_,
        emails0_.id as id1_7_0_,
        emails0_.person_id as person_i2_7_0_
    from
        Email emails0_
    where
        emails0_.person_id in (
            ?, ?, ?
        )
com.p92.javacourse.jee.jpa.associations.Email@4b019895
com.p92.javacourse.jee.jpa.associations.Email@1e835f05
com.p92.javacourse.jee.jpa.associations.Person@43fa0fc6
-- Iterate emails: --
com.p92.javacourse.jee.jpa.associations.Email@5f7e228c
com.p92.javacourse.jee.jpa.associations.Email@55c884ab
com.p92.javacourse.jee.jpa.associations.Person@5621b307
-- Iterate emails: --
com.p92.javacourse.jee.jpa.associations.Email@520e464e
com.p92.javacourse.jee.jpa.associations.Email@2729d4f8
com.p92.javacourse.jee.jpa.associations.Person@3e70bcbe
-- Iterate emails: --
Hibernate:
    /* load one-to-many com.p92.javacourse.jee.jpa.associations.Person.emails */ select
        emails0_.person_id as person_i2_21_1_,
        emails0_.id as id1_7_1_,
        emails0_.id as id1_7_0_,
        emails0_.person_id as person_i2_7_0_
    from
        Email emails0_
    where
        emails0_.person_id in (
            ?, ?, ?
        )
com.p92.javacourse.jee.jpa.associations.Email@3826ea10
com.p92.javacourse.jee.jpa.associations.Email@286bb8a3
com.p92.javacourse.jee.jpa.associations.Person@5b777df8
-- Iterate emails: --
com.p92.javacourse.jee.jpa.associations.Email@5e6cd13c
com.p92.javacourse.jee.jpa.associations.Email@2d0dd3d
com.p92.javacourse.jee.jpa.associations.Person@1ef53b2
-- Iterate emails: --
com.p92.javacourse.jee.jpa.associations.Email@3a1cea92
com.p92.javacourse.jee.jpa.associations.Email@406c350c
com.p92.javacourse.jee.jpa.associations.Person@75fcc1c
```

Subselect fetching strategy

- fetch=”subselect” or @Fetch(FetchMode.SUBSELECT)
- If one lazy collection has to be fetched, Hibernate will load all of them, re-running the original query in a subselect.
- This works in the same way as batch-fetching but without the piecemeal loading.
- Example:

```
public class Person implements Serializable
{
 ...
 @Fetch(FetchMode.SUBSELECT)
 @OneToMany(cascade=CascadeType.ALL,mappedBy="person",fetch=FetchType.LAZY)
 private Set<Email> emails = new HashSet<Email>();
 ...
}
```

```
System.out.println( "-- SELECT person --");
Person person = (Person)session.find(Person.class, p.getId() );
System.out.println( person );

System.out.println( "-- Iterate emails: --");
for ( Email email: person.getEmails()  )
{
 System.out.println( email );
}
```

```
-- SELECT persons --
Hibernate:
    /*
FROM
    Person */ select
        person0_.id as id1_21_
    from
        Person person0_
Iterating persons
com.p92.javacourse.jee.jpa.associations.Person@6454826
-- Iterate emails: --
Hibernate:
    /* load one-to-many com.p92.javacourse.jee.jpa.associations.Person.emails */ select
        emails0_.person_id as person_i2_21_1_,
        emails0_.id as id1_7_1_,
        emails0_.id as id1_7_0_,
        emails0_.person_id as person_i2_7_0_
    from
        Email emails0_
    where
        emails0_.person_id in (
            select
                person0_.id
            from
                Person person0_
        )
com.p92.javacourse.jee.jpa.associations.Email@26041cb3
com.p92.javacourse.jee.jpa.associations.Email@4cde17e5
```

```
com.p92.javacourse.jee.jpa.associations.Person@2e83ba01
-- Iterate emails: --
com.p92.javacourse.jee.jpa.associations.Email@57f83f44
com.p92.javacourse.jee.jpa.associations.Email@6b01b67a
com.p92.javacourse.jee.jpa.associations.Person@4a28af82
-- Iterate emails: --
com.p92.javacourse.jee.jpa.associations.Email@3d733a78
com.p92.javacourse.jee.jpa.associations.Email@5803bbcc
com.p92.javacourse.jee.jpa.associations.Person@351aa5d7
-- Iterate emails: --
com.p92.javacourse.jee.jpa.associations.Email@5d1e7b38
com.p92.javacourse.jee.jpa.associations.Email@7c2bc94a
com.p92.javacourse.jee.jpa.associations.Person@687de17d
-- Iterate emails: --
com.p92.javacourse.jee.jpa.associations.Email@1eac58f6
com.p92.javacourse.jee.jpa.associations.Email@45048e35
com.p92.javacourse.jee.jpa.associations.Person@205f0b23
-- Iterate emails: --
com.p92.javacourse.jee.jpa.associations.Email@2e145cf9
com.p92.javacourse.jee.jpa.associations.Email@3ae3f711
com.p92.javacourse.jee.jpa.associations.Person@3fd4376f
-- Iterate emails: --
com.p92.javacourse.jee.jpa.associations.Email@5f7e228c
com.p92.javacourse.jee.jpa.associations.Email@55c884ab
com.p92.javacourse.jee.jpa.associations.Email@6518740f
com.p92.javacourse.jee.jpa.associations.Email@2b5cf9de
com.p92.javacourse.jee.jpa.associations.Email@520e464e
com.p92.javacourse.jee.jpa.associations.Email@4b019895
com.p92.javacourse.jee.jpa.associations.Email@368a8022
com.p92.javacourse.jee.jpa.associations.Email@4c719b4b
com.p92.javacourse.jee.jpa.associations.Email@286bb8a3
com.p92.javacourse.jee.jpa.associations.Email@2729d4f8
com.p92.javacourse.jee.jpa.associations.Person@5837926
-- Iterate emails: --
com.p92.javacourse.jee.jpa.associations.Email@5e6cd13c
com.p92.javacourse.jee.jpa.associations.Email@3a1cea92
com.p92.javacourse.jee.jpa.associations.Email@6b9ea0a7
com.p92.javacourse.jee.jpa.associations.Email@3826ea10
com.p92.javacourse.jee.jpa.associations.Email@4a640180
com.p92.javacourse.jee.jpa.associations.Email@406c350c
com.p92.javacourse.jee.jpa.associations.Email@7b456b23
com.p92.javacourse.jee.jpa.associations.Email@2d0dd3d
com.p92.javacourse.jee.jpa.associations.Email@a7d060f
com.p92.javacourse.jee.jpa.associations.Email@ec7df
com.p92.javacourse.jee.jpa.associations.Person@7a86b09d
-- Iterate emails: --
com.p92.javacourse.jee.jpa.associations.Email@4324438
com.p92.javacourse.jee.jpa.associations.Email@49cc4e72
com.p92.javacourse.jee.jpa.associations.Email@737b6ad4
com.p92.javacourse.jee.jpa.associations.Email@1cbaaf
com.p92.javacourse.jee.jpa.associations.Email@2ba2bf87
com.p92.javacourse.jee.jpa.associations.Email@34fd48bb
com.p92.javacourse.jee.jpa.associations.Email@1618c82a
com.p92.javacourse.jee.jpa.associations.Email@60b2082a
com.p92.javacourse.jee.jpa.associations.Email@5d474abb
com.p92.javacourse.jee.jpa.associations.Email@48362efe
com.p92.javacourse.jee.jpa.associations.Person@20b55243
-- Iterate emails: --
com.p92.javacourse.jee.jpa.associations.Email@6cc5bd01
```

```
com.p92.javacourse.jee.jpa.associations.Email@4736efa7
com.p92.javacourse.jee.jpa.associations.Email@1ba52c2a
com.p92.javacourse.jee.jpa.associations.Email@4bfb80cf
com.p92.javacourse.jee.jpa.associations.Email@13f01f0
com.p92.javacourse.jee.jpa.associations.Email@67afe177
com.p92.javacourse.jee.jpa.associations.Email@118b2918
com.p92.javacourse.jee.jpa.associations.Email@78f68793
com.p92.javacourse.jee.jpa.associations.Email@5e243737
com.p92.javacourse.jee.jpa.associations.Email@4f983433
com.p92.javacourse.jee.jpa.associations.Person@602bbd7b
-- Iterate emails: --
com.p92.javacourse.jee.jpa.associations.Email@4d4a28ce
com.p92.javacourse.jee.jpa.associations.Email@5c814530
com.p92.javacourse.jee.jpa.associations.Email@31d104d0
com.p92.javacourse.jee.jpa.associations.Email@7f45974f
com.p92.javacourse.jee.jpa.associations.Email@29fbc471
com.p92.javacourse.jee.jpa.associations.Email@3fbd1406
com.p92.javacourse.jee.jpa.associations.Email@41a9fd8e
com.p92.javacourse.jee.jpa.associations.Email@2ae581c2
com.p92.javacourse.jee.jpa.associations.Email@11930d3d
com.p92.javacourse.jee.jpa.associations.Email@501dec3f
com.p92.javacourse.jee.jpa.associations.Person@57a50790
-- Iterate emails: --
com.p92.javacourse.jee.jpa.associations.Email@512d467f
com.p92.javacourse.jee.jpa.associations.Email@33c1da84
com.p92.javacourse.jee.jpa.associations.Email@799a92d1
com.p92.javacourse.jee.jpa.associations.Email@166db69b
com.p92.javacourse.jee.jpa.associations.Email@7cef859d
com.p92.javacourse.jee.jpa.associations.Email@736f657b
com.p92.javacourse.jee.jpa.associations.Email@16277983
com.p92.javacourse.jee.jpa.associations.Email@166fa972
com.p92.javacourse.jee.jpa.associations.Email@7bf12aa7
com.p92.javacourse.jee.jpa.associations.Email@7cdd86df
com.p92.javacourse.jee.jpa.associations.Person@1393abe4
-- Iterate emails: --
com.p92.javacourse.jee.jpa.associations.Email@7409770b
com.p92.javacourse.jee.jpa.associations.Email@1e3c4ea4
com.p92.javacourse.jee.jpa.associations.Email@94f007e
com.p92.javacourse.jee.jpa.associations.Email@11c8ecf3
com.p92.javacourse.jee.jpa.associations.Email@21a954f8
com.p92.javacourse.jee.jpa.associations.Email@2456f37e
com.p92.javacourse.jee.jpa.associations.Email@131d15b6
com.p92.javacourse.jee.jpa.associations.Email@1df44c7a
com.p92.javacourse.jee.jpa.associations.Email@1966aeb1
com.p92.javacourse.jee.jpa.associations.Email@5a7c6987
com.p92.javacourse.jee.jpa.associations.Person@43fa0fc6
-- Iterate emails: --
com.p92.javacourse.jee.jpa.associations.Email@23d469cf
com.p92.javacourse.jee.jpa.associations.Email@59217628
com.p92.javacourse.jee.jpa.associations.Email@6bf5210
com.p92.javacourse.jee.jpa.associations.Email@1b98fdad
com.p92.javacourse.jee.jpa.associations.Email@42b106b9
com.p92.javacourse.jee.jpa.associations.Email@4e6ea769
com.p92.javacourse.jee.jpa.associations.Email@7aa49be5
com.p92.javacourse.jee.jpa.associations.Email@59367702
com.p92.javacourse.jee.jpa.associations.Email@488be7d5
com.p92.javacourse.jee.jpa.associations.Email@74307de3
com.p92.javacourse.jee.jpa.associations.Person@5b777df8
-- Iterate emails: --
```

```
com.p92.javacourse.jee.jpa.associations.Email@4b936059
com.p92.javacourse.jee.jpa.associations.Email@4cf7164a
com.p92.javacourse.jee.jpa.associations.Email@5f33f2f3
com.p92.javacourse.jee.jpa.associations.Email@71d87bbf
com.p92.javacourse.jee.jpa.associations.Email@d627276
com.p92.javacourse.jee.jpa.associations.Email@3ba472fd
com.p92.javacourse.jee.jpa.associations.Email@3a48a7d7
com.p92.javacourse.jee.jpa.associations.Email@2de8c706
```

```
com.p92.javacourse.jee.jpa.associations.Email@389a99d2
com.p92.javacourse.jee.jpa.associations.Email@29795ea2
com.p92.javacourse.jee.jpa.associations.Person@1ef53b2
```

Associtation mapping and Collection Performance

Problems

- Unexpected SQL statements
- Too many SQL statements
- Too many objects are loaded into memory

Inadequate collection type on the owning side (@OneToMany)

- https://fedcsis.org/proceedings/2013/pliks/322.pdf
- Bag
  - Always recreates the collection.
- List
  - It needs to update the indices.
- Set
  - Nothing special
- Number of DML operations issued while persisting.

Semantics	One Element Added	One Element Removed	Java Type
Bag semantics	1 delete, N inserts	1 delete, N inserts	java.util.Collection java.util.List
List semantics	1 insert, M updates	1 insert, M updates	java.util.List
Set semantics	1 insert	1 delete	java.util.Set

- Recommendation
  - No changes in the collection / minimal change / relative small changes -> java.util.Set with set semantics
  - The collection is heavily modified -> java.util.Collection or java.util.List with bag semantics

OneToMany as owning side

- Description:
  - usage of the collection side as the owning side of an association for large collections especially where only a few changes in a single transaction.
- Problems:
  - The entire collection is loaded into memory
    - -> large memory footprint -> BAD
    - -> uniqueness check in the collection -> large computation power -> BAD
    - -> large amount of objects are created from records -> large computation power -> BAD
  - With optimistic locking we lock the entity and the entities in the collection -> lower the concurrent access capabilities -> BAD
- Solution
  - Make the @ManyToOne of the owning side of the association.
  - If we can't do this exclude it from optimistic locking with @OptimisticLock(excluded=true)

Lost collection proxy on the owning side

- Description:
  - Assignment of a new collection object to a persistent field representing the owning side of a one-to-many

association.

- Prolem:
  - The collection proxy lost -> Hibernate will recreate the whole collection (DML delete and N DML insert)
- Solution:
  - Few changes -> work on the proxy collection
  - Many changes -> assign a new collection

Performance monitoring

Monitoring the SessionFactory

- SessionFactory#getStatistics()
  - https://docs.jboss.org/hibernate/orm/4.2/javadocs/org/hibernate/SessionFactory.html#getStatistics()
  - It returns a Statistics object. https://docs.jboss.org/hibernate/orm/4.2/javadocs/org/hibernate/stat/Statistics.html
  - The metrics are platform dependents.
- You can enable the StatisticsService as MBean.
  - You can enable one mbean / factory.

```
Hashtable tb = new Hashtable();
tb.put("type", "statistics");
tb.put("sessionFactory", "myFinancialApp");
ObjectName on = new ObjectName("hibernate", tb);
StatisticsService stats = new StatisticsService();
stats.setSessionFactory(sessionFactory);
server.registerMBean(stats, on);
```

  - You can enable one mben for all factories.

```
Hashtable tb = new Hashtable();
tb.put("type", "statistics");
tb.put("sessionFactory", "myFinancialApp");
ObjectName on = new ObjectName("hibernate", tb);
StatisticsService stats = new StatisticsService();
server.registerMBean(stats, on);
```

  - You can enable/disable the monitoring of the SessionFactory via
    - setting the hibernate.generate_statistics to true/false at configuration time.
    - or invoking the SessionFactory.getStatistics().setStatisticsEnabled(boolean) method.

Monitoring the Session

- Use the org.hibernate.engine.internal.StatisticalLoggingSessionEventListener.
- https://docs.jboss.org/hibernate/orm/4.2/javadocs/org/hibernate/engine/internal/StatisticalLoggingSessionEventListener.html
- It is automatically registered and logs the statistics according to your logging settings.
- Enable logging by setting the property hibernate.generate_statistics to true. -> Hibernate will write one multi-line log statement with summarized statistical information at the end of the session
- Do NOT activate it on production system.
- Set the following system properties to true to log sql statements:
  - hibernate.show_sql
  - hibernate.format_sql

Example Session log:

```
okt. 28, 2015 6:45:51 DU org.hibernate.engine.internal.StatisticalLoggingSessionEventListener end
INFO: Session Metrics {
    3843 nanoseconds spent acquiring 1 JDBC connections;
    0 nanoseconds spent releasing 0 JDBC connections;
    607530 nanoseconds spent preparing 17 JDBC statements;
    10052056 nanoseconds spent executing 17 JDBC statements;
    0 nanoseconds spent executing 0 JDBC batches;
    0 nanoseconds spent performing 0 L2C puts;
    0 nanoseconds spent performing 0 L2C hits;
    0 nanoseconds spent performing 0 L2C misses;
    26309267 nanoseconds spent executing 1 flushes (flushing a total of 469 entities and 47
collections);
    54182 nanoseconds spent executing 1 partial-flushes (flushing a total of 0 entities and 0 collections)
}
```

Original ToC

1. Persistence context
2. Mappings
3. Fetching
4. Native queries
5. Report queries
6. Batching
7. Caching
8. Transactions (hibák logolása, kezelése, rollback is)
9. Locking
10. Profiling and debugging
11. Indexing
12. Data type tuning és Dialect

Notes:

- A very good performance tutorial:
  - http://vladmihalcea.com/tutorials/hibernate/
  - Code: https://github.com/vladmihalcea/hibernate-master-class
- Collection performance:
  - http://docs.jboss.org/hibernate/orm/4.3/manual/en-US/html/ch20.html#performance-collections-mostefficentinverse
- Paging in hibernate
  - https://erikwramner.wordpress.com/2013/11/18/paging-gotcha-with-hibernate-jpa/
- Jboss performance tuning guide
  - https://access.redhat.com/documentation/en-US/JBoss_Enterprise_Application_Platform/5/html/Performance_Tuning_Guide/sect-Performance_Tuning_Guide-Entity_Beans-Batching_Database_Operations.html
- select-before-update
- http://www.slideshare.net/brmeyer/hibernate-orm-performance-31550150
- http://stackoverflow.com/questions/3144163/with-spring-transactions-with-hibernate-how-can-you-get-2000-inserts-to-not-sl
- Write behind and write through caches:
  - http://www.ehcache.org/generated/2.10.0/html/ehc-all/index.html#page/Ehcache_Documentation_Set/co-write_about_write_through_behind.html
  - http://www.infoq.com/articles/write-behind-caching
- Dirty checking
  - http://vladmihalcea.com/2014/08/21/the-anatomy-of-hibernate-dirty-checking/
  - http://vladmihalcea.com/2014/08/29/how-to-customize-hibernate-dirty-checking-mechanism/
  - http://prismoskills.appspot.com/lessons/Hibernate/Chapter_20_-_Dirty_checking.jsp
- Performance antipatterns

    - https://fedcsis.org/proceedings/2013/pliks/322.pdf
- Identifier generators
    - http://vladmihalcea.com/2014/06/23/the-hilo-algorithm/
    - http://vladmihalcea.com/2014/07/15/from-jpa-to-hibernates-legacy-and-enhanced-identifier-generators/
    - http://vladmihalcea.com/2014/07/21/hibernate-hidden-gem-the-pooled-lo-optimizer/
- http://www.javacodegeeks.com/2014/06/performance-tuning-of-springhibernate-applications.html
- Query Cache
    - http://apmblog.dynatrace.com/2009/02/16/understanding-caching-in-hibernate-part-two-the-query-cache/
    - http://darren.oldag.net/2008/11/hibernate-query-cache-dirty-little_04.html
- Second level cache
    - http://richardchesterwood.blogspot.hu/2013/01/configuring-cacheconcurrencystrategy-in.html

fetch=”join” or @Fetch(FetchMode.JOIN)