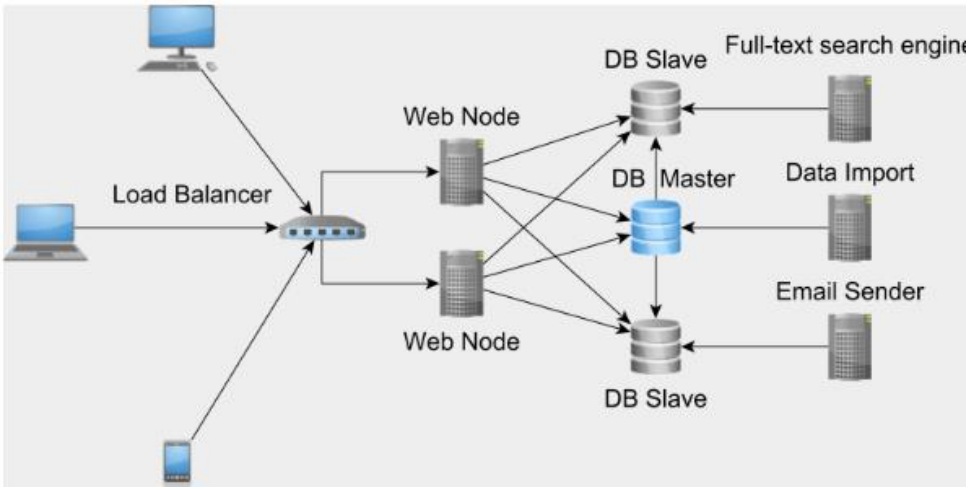
HIBERNATE GOLDEN RULES (https://vladmihalcea.com/tutorials/hibernate/)



1. Hibernate defines the hibernate.show_sql configuration property to enable logging.
Because Hibernate uses PreparedStatement(s) exclusively, the bind parameter are not visible.
So, use the DataSource-proxy to see the bind parameters.

---

2. A testing-time assertion mechanism is even better because you can catch N+1 query problems
even before you commit your code. This will allow you to test if the number of executed
queries is different from the number of expected queries to be executed.

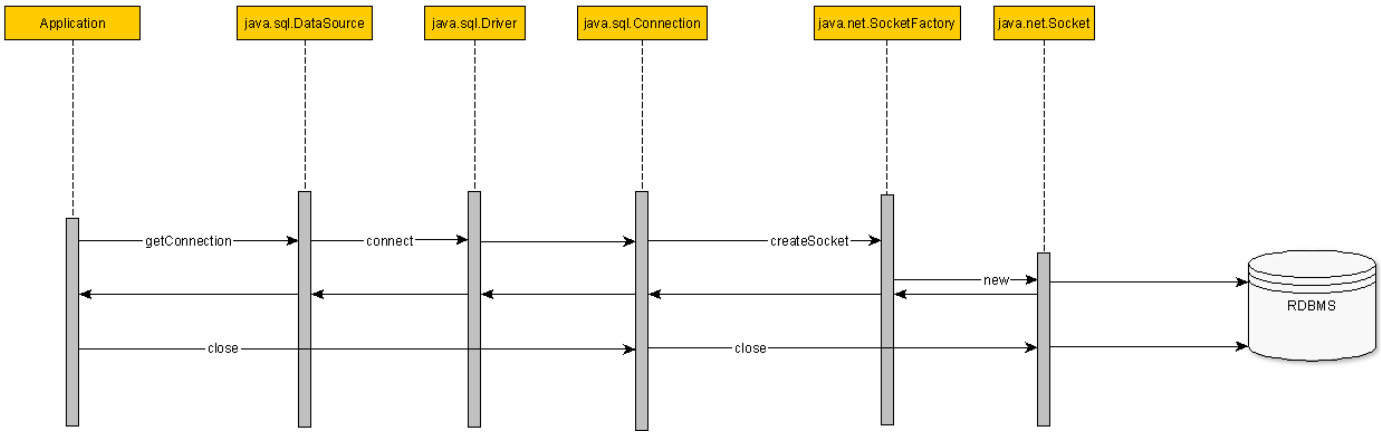
---

3. Database connections are expensive, therefore you should always use a connection pooling mechanism.
The fastest connection pool is HikariCP.

---

4. The database connection life-cycle flow is:

https://vladmihalcea.files.wordpress.com/2014/04/connectionlifecycle.gif
The database connection life-cycle

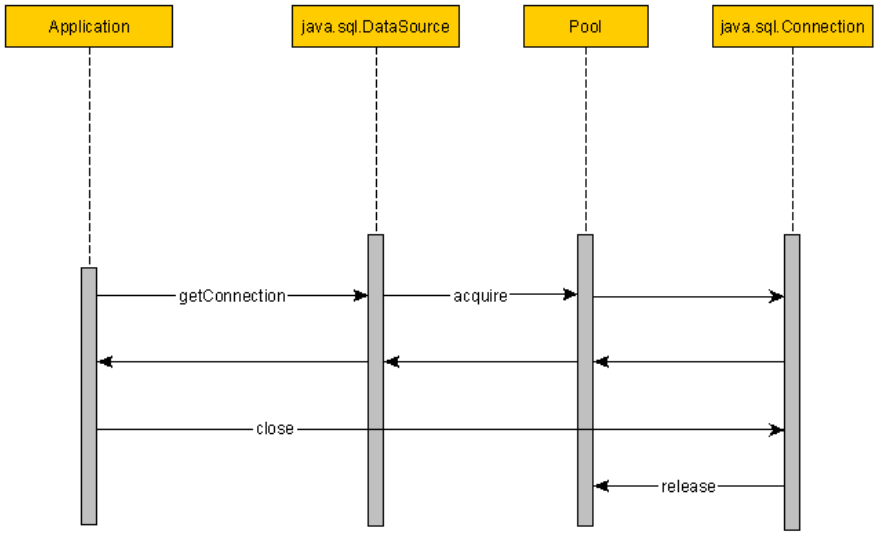


a) The application data layer ask the DataSource for a database connection
b) The DataSource will use the database Driver to open a database connection
c) A database connection is created and a TCP socket is opened
d) The application reads/writes to the database
e) The connection is no longer required so it is closed
f) The socket is closed

---

5. Commonly, for a connection pool we need to configure the following: minimum size, maximum size,
max idle time, acquire timeout, timeout and retry attempts.

https://vladmihalcea.files.wordpress.com/2014/04/poolingconnectionlifecycle.gif
Pooling connection management flow



---

6. JDBC batching allows us to send multiple SQL statements in a single database roundtrip. The performance gain is significant both on the Driver and the database side. PreparedStatements are very good candidates for batching, and some database systems (e.g. Oracle) support batching only for prepared statements only.

---

7. To enable JDBC batching, we have to configure the hibernate.jdbc.batch_size property. A non-zero value enables use of JDBC2 batch updates by Hibernate (e.g. recommended values between 5 and 30).

---

8. The default batch size can be obtained via the org.hibernate.dialect.Dialect.DEFAULT_BATCH_SIZE

---

9. Hibernate 5.2 offers Session-level batching, so it's even more flexibile in this regard. This is done via: Session#setJdbcBatchSize(int size) and Session#getJdbcBatchSize().

---

10. Statement caching is one of the least-known performance optimization that you can easily take advantage of. Depending on the underlying JDBC Driver, you can cache PreparaedStatements both on the client-side (the Driver) or databases-side (either the syntax tree or even the execution plan).

---

11. When using Hibernate, the IDENTITY generator is not a good choice since it disables JDBC batching.

---

12. TABLE generator is even worse since it uses a separate transaction for fetching a new identifier, which can put pressure on the underlying transaction log, as well as the connection pool since a separate connection is required every time we need a new identifier.

---

13. SEQUENCE is the right choice, and even SQL Server supports since version 2012. For SEQUENCE identifiers, Hibernate has long been offering optimizers like pooled or pooled-lo which can reduce the number of database roundtrips required for fetching a new entity identifier value.

---

14. You should always use the right column types on the database side. The more compact the column type is, the more entries can be accommodated in the database working set, and indexes will better fit into memory. For this purpose, you should take advantage of database-specific types (e.g. inet for IPv4 addresses in PostgreSQL), especially since Hibernate is very flexible when it comes to implementing a new custom Type.

---

15. Unidirectional associations and @ManyToMany should always be avoided. For collection, bidirectional @OneToMany associations are preferred. However, unlike queries, collections are less flexible since they cannot be easily paginated, meaning that we cannot use them when the number of child associations is rather high. For this reason, you should always question if a collection is really necessary. An entity query might be a better alternative in many situations.

---

16. When it comes to inheritance, the impedance mismatch between object-oriented languages and relational databases becomes even more apparent. JPA offers SINGLE_TABLE, JOIN, and TABLE_PER_CLASS to deal with inheritance mapping, and each of these strategies has pluses and minuses.

---

17. SINGLE_TABLE performs the best in terms of SQL statements, but we lose on the data integrity side since we cannot use NOT NULL constraints.

---

18. JOIN addresses the data integrity limitation while offering more complex statements. As long as you don't use polymorphic queries or @OneToMany associations against base types, this strategy is fine. Its true power comes from polymorphic @ManyToOne associations backed by a Strategy pattern on the data access layer side.

---

19. TABLE_PER_CLASS should be avoided since it does not render efficient SQL statements.

---

20. When using JPA and Hibernate, you should always mind the Persistence Context size. For this reason, you should never bloat it with tons of managed entities. By restricting the number of managed entities, we gain better memory management, and the default dirty checking mechanism is going to be more efficient as well.

---

21. Fetching too much data is probably the number one cause for data access layer performance issues. One issue is that entity queries are used exclusively, even for read-only projections.

---

22. DTO projections are better suited for fetching custom views, while entities should only be fetched when the business flow requires to modify them.

---

23. EAGER fetching is the worst, and you should avoid anti-patterns such as Open-Session in View.

---

24. Relational database systems use many in-memory buffer structures to avoid disk access. Database caching is very often overlooked. We can lower response time significantly by properly tuning the database engine so that the working set resides in memory and is not fetched from disk all the time.

---

25. Application-level caching is not optional for many enterprise application. Application-level caching can reduce response time while offering a read-only secondary store for when the database is down for maintenance or because of some serious system failure.

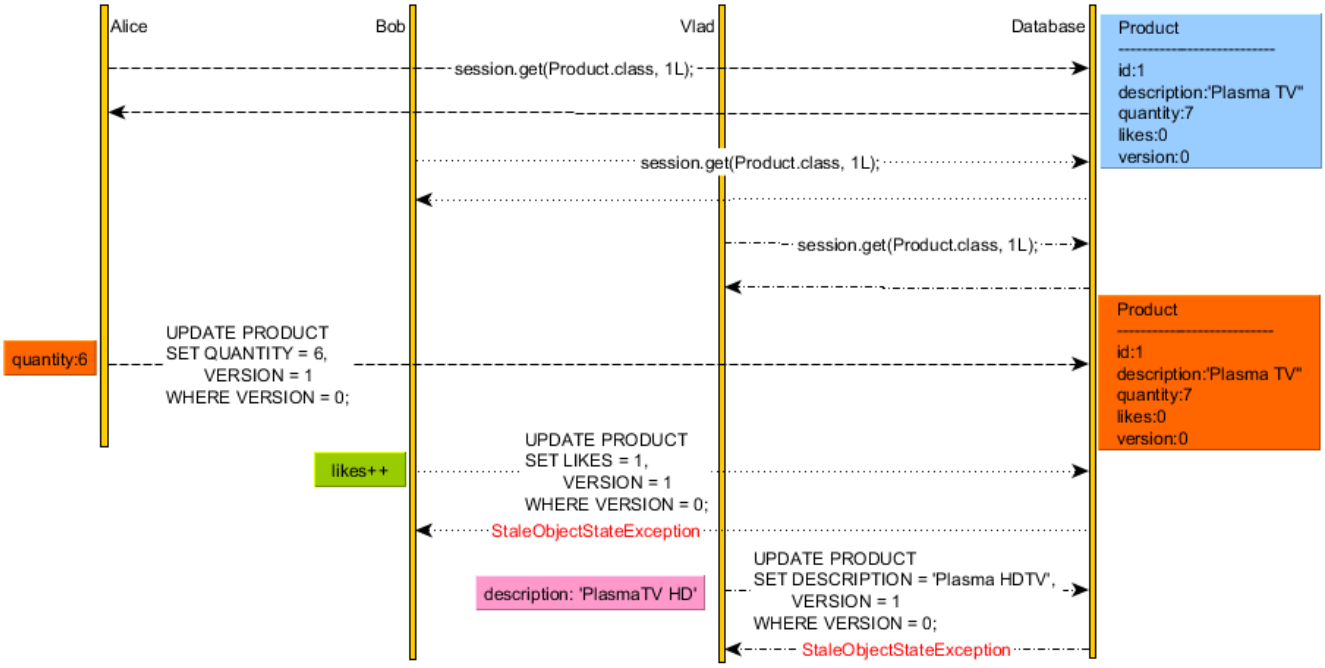
---

26. The second-level cache is very useful for reducing read-write transaction response time, especially in Master-Slave replication architectures. Depending on application requirements, Hibernate allows you to choose between READ_ONLY, NONSTRICT_READ_WRITE, READ_WRITE, and TRANSACTIONAL.

---

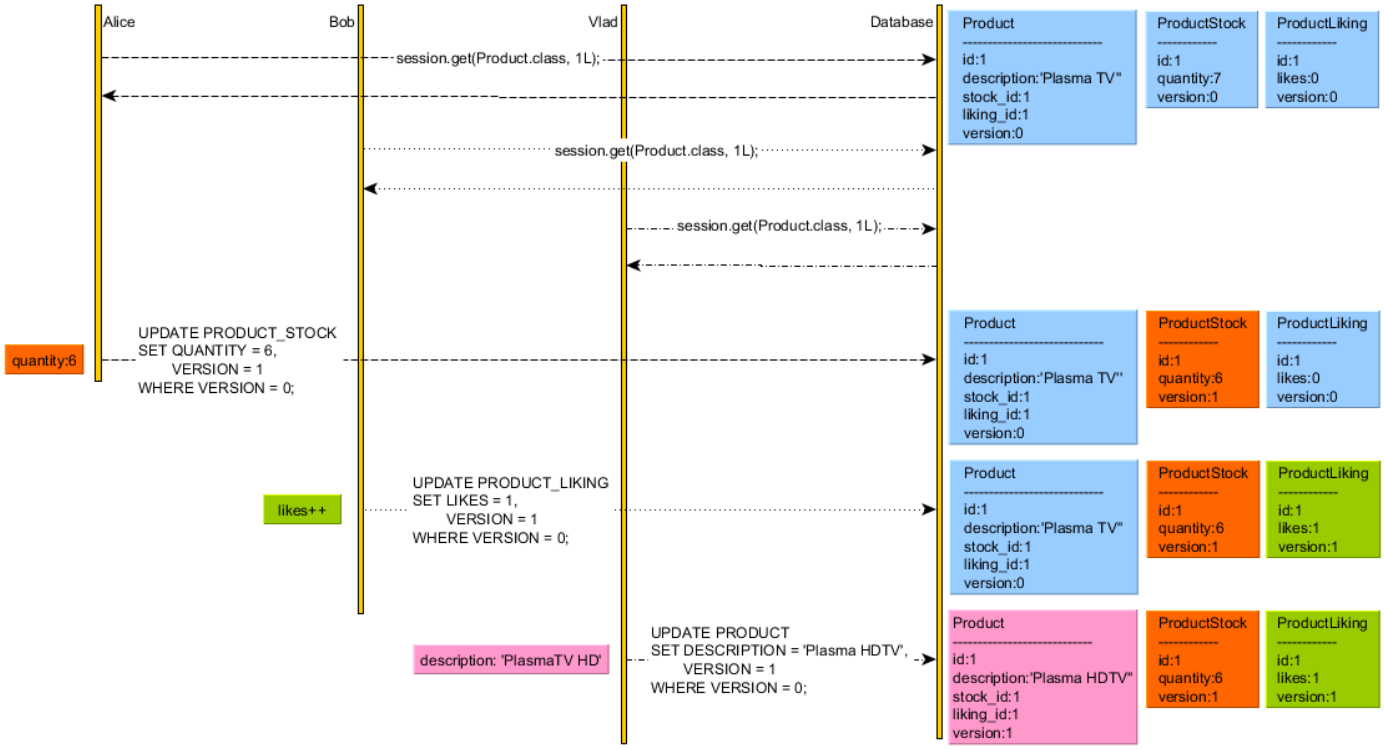
27. The choice of transaction isolation level is of paramount importance when it comes to performance and data integrity. For multi-request web flows, to avoid lost updates, you should use optimistic locking with detached entities or an EXTENDED Persistence Context.

https://vladmihalcea.files.wordpress.com/2014/11/optimisticlockingonerootentityoneversion1.png



After using optimistic locking all three transaction passed successfully.

https://vladmihalcea.files.wordpress.com/2014/11/optimisticlockingonerootentitymultipleversions.png



---

28. To avoid optimistic locking false positives, you can use versionless optimistic concurrency control or split entities based write-based property sets.

---

29. Just because you use JPA or Hibernate, it does not mean that you should not use native queries. You should take advantage of Window Functions, CTE (Common Table Expressions), CONNECT BY, PIVOT.

---

30. These constructs allow you to avoid fetching too much data just to transform it later in the application layer. If you can let the database do the processing, you can fetch just the end result, therefore, saving lots of disk I/O and networking overhead. To avoid overloading the Master node, you can use database replication and have multiple Slave nodes available so that data-intensive tasks are executed on a Slave rather than on the Master.

---

31. Database replication and sharding are very good ways to increase throughput, and you should totally take advantage of these battle-tested architectural patterns to scale your enterprise application.

---

32. Criteria API is very useful for dynamically building queries, but that's the only use case where I'd use it. Whenever you have an UI with N filters that may arrive in any M combinations, it makes sense to have an API to construct queries dynamically, since concatenating strings is always a path I'm running away from.

---

33. Pay attention to SQL queries generated behind the Criteria API!

---

34. Usually in a one-to-one association, one table will contain a PK and a FK. But, since there can be a single record from table A that corresponds to a single record from table B, we can use the PK as FK also and the two tables are sharing their PKs as well.

---

35. PK and FK columns are most often indexed, so sharing the PK can reduce the index footprint by half, which is desirable since you want to store all your indexes into memory to speed up index scanning.

---

36. While the unidirectional @OneToOne association can be fetched lazily, the parent-side of a bidirectional @OneToOne association is not. Even when specifying that the association is not optional and we have the FetchType.LAZY, the parent-side association behaves like a FetchType.EAGER relationship. And EAGER fetching is bad.

---

37. Even if the FK is NOT NULL and the parent-side is aware about its non-nullability through the optional attribute (e.g. @OneToOne(mappedBy = "post", fetch = FetchType.LAZY, optional = false)), Hibernate still generates a secondary select statement.

---

38. Bytecode enhancement is the only viable workaround. However, it only works if the parent side is annotated with @LazyToOne(LazyToOneOption.NO_PROXY) and the child side is not using @MapsId.

---

39. The best way to map a @OneToOne relationship is to use @MapsId. This way, you don't even need a bidirectional association since you can always fetch the FooDetails entity by using the Foo entity identifier.

---

40. This way, the id column serves as both Primary Key and FK. You'll notice that the @Id column no longer uses a @GeneratedValue annotation since the identifier is populated with the identifier of the post association.

---

41. Hibernate shifts the developer mindset from SQL statements to entity state transitions. Once an entity is actively managed by Hibernate, all changes are going to be automatically propagated to the database.

---

42. TRANSIENT - A newly created object that hasn't ever been associated with a Hibernate Session (a.k.a Persistence Context) and is not mapped to any database table row is considered to be in the New (Transient) state. To become persisted we need to either explicitly call the EntityManager#persist() method or make use of the transitive persistence mechanism.

---

43. PERSISTENCE - A persistent entity has been associated with a database table row and it's being managed by the current running Persistence Context. Any change made to such entity is going to be detected and propagated to the database (during the Session flush-time). With Hibernate, we no longer have to execute INSERT/UPDATE/DELETE statements. Hibernate employs a transactional write-behind working style and changes are synchronized at the very last responsible moment, during the current Session flush-time.

---

44. DETACHED - Once the current running Persistence Context is closed all the previously managed entities become detached. Successive changes will no longer be tracked and no automatic database synchronization is going to happen.

    To associate a detached entity to an active Hibernate Session, you can choose one of the following options:

    Reattaching - Hibernate (but not JPA 2.1) supports reattaching through the Session#update method. A Hibernate Session can only associate one Entity object for a given database row. This is because the Persistence Context acts as an in-memory cache (first level cache) and only one value (entity) is associated to a given key (entity type and database identifier).

    An entity can be reattached only if there is no other JVM object (matching the same database row) already associated to the current Hibernate Session.

    Merging - The merge operaration is going to copy the detached entity state (source) to a managed entity instance (destination). If the merging entity has no equivalent in the current Session, one will be fetched from the database.

    The detached object instance will continue to remain detached even after the merge operation.

---

45. Removed - Although JPA demands that managed entities only are allowed to be removed, Hibernate can also delete detached entities (but only through a Session#delete method call). A removed entity is only scheduled for deletion and the actual database DELETE statement will be executed during Session flush-time.

---

46. There's no difference between calling persist(), merge() or refresh() on the JPA EntityManager or the Hibernate Session.

---

47. The JPA remove and detach calls are delegated to Hibernate delete and evict native operations.

---

48. Only Hibernate supports replicate() and saveOrUpdate(). While replicate is useful for some very specific scenarios (when the exact entity state needs to be mirrored between two distinct DataSources), the persist and merge combo is always a better alternative than the native saveOrUpdate operation.

---

49. As a rule of thumb, you should always use persist for TRANSIENT entities and merge for DETACHED ones.

---

50. The JPA lock method shares the same behaviour with Hibernate lock request method.

---

51. The JPA CascadeType.ALL doesn't only apply to EntityManager state change operations, but to all Hibernate CascadeTypes as well.

---

52. So if you mapped your associations with CascadeType.ALL, you can still cascade Hibernate specific events. For example, you can cascade the JPA lock operation (although it behaves as reattaching,

instead of an actual lock request propagation), even if JPA doesn't define a CascadeType.LOCK.

---

53. Cascading only makes sense only for Parent – Child associations (the Parent entity state transition being cascaded to its Child entities). Cascading from Child to Parent is not very useful and usually, it's a mapping code smell.

---

54. Most common one-to-one bidirectional association:

In Foo (parent):
```
@OneToOne(mappedBy = "foo", cascade = CascadeType.ALL, orphanRemoval = true)
private FooDetails details;
```

In FooDetails (child):
```
@OneToOne
@JoinColumn(name = "id")
@MapsId
private Foo foo;
```

---

55. The bidirectional associations should always be updated on both sides, therefore the Parent side should contain the addChild() and removeChild() combo. These methods ensure we always synchronize both sides of the association, to avoid object or relational data corruption issues.

In Foo (parent):
```
public void addDetail(FooDetail detail) {
 this.details.add(detail);
 detail.setFoo(this);
}

public void removeDetail(FooDetail detail) {
  this.details.remove(detail);
 detail.setFoo(this);
}
```

---

56. Cascading the one-to-one persist operation
The CascadeType.PERSIST comes along with the CascadeType.ALL configuration, so we only have to persist the Foo entity, and the associated FooDetails entity is persisted as well.

---

57. Cascading the one-to-one merge operation
The CascadeType.MERGE is inherited from the CascadeType.ALL setting, so we only have to merge the Foo entity and the associated FooDetails is merged as well.

---

58. Cascading the one-to-one delete operation
The CascadeType.REMOVE is also inherited from the CascadeType.ALL configuration, so the Foo entity deletion triggers a FooDetails entity removal too.

---

59. The one-to-one delete orphan cascading operation
If a Child entity is dissociated from its Parent, the Child Foreign Key is set to NULL.
If we want to have the Child row deleted as well, we have to use the orphan removal support.

---

60. Unidirectional one-to-one association
Most often, the Parent entity is the inverse side (e.g. mappedBy), the Child controlling the association through its Foreign Key. But the cascade is not limited to bidirectional associations, we can also use it for unidirectional relationships.

---

61. Cascading consists in propagating the Parent entity state transition to one or more Child entities, and it can be used for both unidirectional and bidirectional associations.

---

62. The most common Parent – Child association consists of a one-to-many and a many-to-one relationship, where the cascade being useful for the one-to-many side only.

In Foo (parent):
```
@OneToMany(cascade = CascadeType.ALL, mappedBy = "foo", orphanRemoval = true)
private List<Buzz> buzzs = new ArrayList<>();

public void addBuzz(Buzz buzz) {
 buzzs.add(buzz);
 buzz.setFoo(this);
}

public void removeBuzz(Buzz buzz) {
 buzz.setFoo(null);
 this.buzzs.remove(buzz);
```

```
}
```

In Buzz (child):

```
@ManyToOne
private Foo foo;
```

Like in the one-to-one example, the CascadeType.ALL and orphan removal are suitable because the Buzz life-cycle is bound to that of its Foo Parent entity.

---

63. Cascading the one-to-many persist operation
We only have to persist the Foo entity and all the associated Buzz entities are persisted also.

---

64. Cascading the one-to-many merge operation
Merging the Foo entity is going to merge all Buzz entities as well.

---

65. Cascading the one-to-many delete operation
When the Foo entity is deleted, the associated Buzz entities are deleted as well.

---

66. The one-to-many delete orphan cascading operation
The orphan-removal allows us to remove the Child entity whenever it's no longer referenced by its Parent.

---

67. The many-to-many relationship is tricky because each side of this association plays both the Parent and the Child role. Still, we can identify one side from where we'd like to propagate the entity state changes.

In Foo (parent and child):

```
@ManyToMany(mappedBy = "foos", cascade = {CascadeType.PERSIST, CascadeType.MERGE})
private List<Buzz> buzzs = new ArrayList<>();

public void addBuzz(Buzz buzz) {
 buzzs.add(buzz);
 buzz.foos.add(this);
}

public void removeBuzz(Buzz buzz) {
 buzzs.remove(buzz);
 buzz.foos.remove(this);
}

public void remove() {
 for(Buzz buzz : new ArrayList<>(buzzs)) {
     removeBuzz(buzz);
 }
}
```

In Buzz (parent and child):

```
@ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE})
@JoinTable(name = "Buzz_Foo", joinColumns = { @JoinColumn(
           name = "buzz_id", referencedColumnName = "id")
}, inverseJoinColumns = { @JoinColumn( name = "foo_id",  referencedColumnName = "id")
})
private List<Foo> foos = new ArrayList<>();
```

---

68. We shouldn't default to CascadeType.ALL, because the CascadeTpe.REMOVE might end-up deleting more than we're expecting.

---

69. Cascading the many-to-many persist operation
Persisting the Foo entities will persist the Buzz as well.

---

70. Dissociating one side of the many-to-many association
To delete an Foo, we need to dissociate all Foo_Buzz relations belonging to the removable entity.
Delete all Buzz via Foo#remove() and afterwards call Session#delete(foo).

---

71. The many-to-many CascadeType.REMOVE gotchas

The many-to-many CascadeType.ALL is another code smell, I often bump into while reviewing code. The CascadeType.REMOVE is automatically inherited when using CascadeType.ALL, but the entity removal is not only applied to the link table, but to the other side of the association as well. Let's change the Foo entity books many-to-many association to use the CascadeType.ALL instead:

In Foo (parent and child):

```
@ManyToMany(mappedBy = "foos", cascade = CascadeType.ALL)
```

```
private List<Buzz> buzzs = new ArrayList<>();
```

When deleting one Foo, all Buzz belonging to the deleted Foo are getting deleted, even if other fOO we're still associated to the deleted Buzz. Most often, this behavior doesn't match the business logic expectations, only being discovered upon the first entity removal.

---

72. We can push this issue even further, if we set the CascadeType.ALL to the Buzz entity side as well. This time, not only the Buzz are being deleted, but Foo are deleted as well.

---

73. That's why CascadeType.ALL should raise your eyebrow, whenever you spot it on a many-to-many association.

---

74. Practical test cases for real many-to-many associations are rare. Most of the time you need additional information stored in the link table. In this case, it is much better to use two one-to-many associations to an intermediate link class. In fact, most associations are one-to-many and many-to-one. For this reason, you should proceed cautiously when using any other association style.

---

75. Speaking about the @ElementCollection.

```
@ElementCollection
@CollectionTable(
 name="foo_change",
 joinColumns=@JoinColumn(name="foo_id")
)
private List<Change> changes = new ArrayList<>();
```

The Change object is modeled as an Embeddable type and it can only be accessed through its owner Entity. The Embeddable has no identifier and it cannot be queried through JPQL. The Embeddable life-cycle is bound to that of its owner, so any Entity state transition is automatically propagated to the Embeddable collection.

By default, any collection operation ends up recreating
the whole data set. This behavior is only acceptable for an in-memory collection and it's not suitable from a database perspective. The database has to delete all existing rows, only to re-add them afterwords. The more indexes we have on this table, the greater the performance penalty.

---

76. Because an Embeddable cannot contain an identifier, we can at least add an order column so that each row can be uniquely identified. Let's see what happens when we add an @OrderColumn to our element collection:

```
@ElementCollection
@CollectionTable(
 name="foo_change",
 joinColumns=@JoinColumn(name="foo_id")
)
@OrderColumn(name = "index_id")
private List<Change> changes = new ArrayList<>();
```

And, put nullable = false to Change columns:

```
@Column(name = "foo", nullable = false)
private String foo;

@Column(name = "buzz", nullable = false)
private String buzz;
```

---

76. Compared to an inverse one-to-many association, the ElementCollection is more difficult to optimize. If the collection is frequently updated then a collection of elements is better substituted by a one-to-many association. Element collections are more suitable for data that seldom changes, when we don't want to add an extra Entity just for representing the foreign key side.

---

77. When it comes to controlling the persist/merge part of the association, there are two options available. One would be to have the @OneToMany end in charge of synchronizing the collection changes, but this is an inefficient approach. The most common approach is when the @ManyToOne side controls the association and the @OneToMany end is using the "mappedBy" option.

---

79. So, for bidirectional collections, we could use a java.util.List or a java.util.Set. When using List pay attention to HHH-5855 - https://hibernate.atlassian.net/browse/HHH-5855.
The HHH-5855 issue was fixed in Hibernate 5.0.8, so another reason to update.
This issue only replicates if a merge operations is cascaded from parent to children.

---

78. One advantage of using Sets is that it forces you to define a proper equals/hashCode strategy (which should always include the entity's business key. A business key is a field combinations that's unique, or unique among a parent's children, and that's consistent even before and after the entity is persisted into the database).

---

79. If you are worried you are going to lose the List ability of saving the children in the same order you've added them, then you can still emulate this for Sets too. By default Sets are unordered and unsorted, but even if you can't order them you may still sort them by a given column, by using the @OrderBy JPA annotation:

```
@OneToMany(cascade = CascadeType.ALL, mappedBy = "foo", orphanRemoval = true)
@OrderBy("id")
private Set children = new LinkedHashSet();
```

But if you need duplicates you can still use an Indexed List. So, I'm still cautious about Bags, and if my domain model imposes using a List I'd always pick the indexed one.

---

80. When choosing a primary key we must take into consideration the following aspects:

- the primary key may be used for joining other tables through a foreign key relationship
- the primary key usually has an associated default index, so the more compact the data type the less space the index will take
- a simple key performs better than a compound one
- the primary key assignment must ensure uniqueness even in highly concurrent environments

---

81. When choosing a primary key generator strategy the options are:

- natural keys, using a column combination that guarantees individual rows uniqueness
- surrogate keys, that are generated independently of the current row data

---

82. Natural keys' uniqueness is enforced by external factors (e.g. person unique identifiers, social security numbers, vehicle identification numbers). Natural keys are convenient because they have an outside world equivalent and they don't require any extra database processing. We can therefore know the primary key even before inserting the actual row into the database, which simplifies batch inserts. If the natural key is a single numeric value the performance is comparable to that of surrogate keys.

---

83. For compound keys we must be aware of possible performance penalties:

- compound key joins are slower than single key ones
- compound key indexes require more space than their single key counterparts

---

84. Non-numerical keys are less efficient than numeric ones (integer, bigint), for both indexing and joining. A CHAR(17) natural key (e.g. vehicle identification number) occupies 17 bytes as opposed to 4 bytes (32 bit integer) or 8 bytes (64 bit bigint).

---

85. Surrogate keys are generated independently of the current row data, so the other column constraints may freely evolve according to the application business requirements. The database system may manage the surrogate key generation and most often the key is of a numeric type (e.g. integer or bigint), being incremented whenever there is a need for a new key.

---

86. If we want to control the surrogate key generation we can employ a 128-bit GUID or UUID. This simplifies batching and may improve the insert performance since the additional database key generation processing is no longer required. Even if this strategy is not so widely adopted it's worth considering when designing the database model.

---

87. When the database identifier generation responsibility falls to the database system, there are several strategies for auto incrementing surrogate keys:

Database engine	**Auto incrementing strategy**
Oracle	SEQUENCE, IDENTITY (Oracle 12c)
MSSQL	IDENTITY, SEQUENCE (MSSQL 2012)
PostgreSQL	SEQUENCE, SERIAL TYPE
MySQL	AUTO_INCREMENT
DB2	IDENTITY, SEQUENCE
HSQLDB	IDENTITY, SEQUENCE

---

88. Because sequences may be called concurrently from different transactions they are usually transaction-less.

- Oracle When a sequence number is generated, the sequence is incremented, independent of the

transaction committing or rolling back
- MSSQL Sequence numbers are generated outside the scope of the current transaction. They are consumed whether the transaction using the sequence number is committed or rolled back
- PostgreSQL Because sequences are non-transactional, changes made by setval are not undone if the transaction rolls back

---

89. Both the IDENTITY type and the SEQUENCE generator are defined by the SQL:2003 standard, so they've become the standard primary key generator strategies.
Some database engines allow you to choose between IDENTITY and SEQUENCE so you have to decide which one better suits your current schema requirements.
Hibernate disables JDBC insert batching when using the IDENTITY generator strategy.

---

90. But even with cached sequences, the application requires a database round-trip for every new the sequence value. If your applications demand a high number of insert operations per transaction, the sequence allocation may be optimized with a hi/lo algorithm.

---

91. Hibernate offers many identifier strategies to choose from and for UUID identifiers

---

92. From JPA to Hibernate flushing strategies

jpa	**hibernate**	
AUTO	AUTO	The Session is sometimes flushed before query execution.
COMMIT	COMMIT	The Session is only flushed prior to a transaction commit.
ALWAYS		The Session is always flushed before query execution.
MANUAL		The Session can only be manually flushed.

---

93. The Persistence Context defines a default flush mode, that can be overridden upon Hibernate Session creation. Queries can also take a flush strategy, therefore overruling the current Persistence Context flush mode.

Scope	**Hibernate**	**JPA**
Persistence Context	Session	EntityManager
Query	Query	Query
	Criteria	TypedQuery

---

94. The persist operation must be used only for new entities. From JPA perspective, an entity is new when it has never been associated with a database row, meaning that there is no table record in the database to match the entity in question.

---

95. The TABLE strategy behaves like SEQUENCE, but you should avoid it at any cost because it uses a separate transaction to generate the entity identifier, therefore putting pressure on the underlying connection pool and the database transaction log.
Even worse, row-level locks are used to coordinate multiple concurrent requests, and, just like Amdhal's Law tells us, introducing a serializability execution can affect scalability.

---

96. While for IDENTITY and SEQUENCE generator strategies, you can practically use merge to persist an entity, for the assigned generator, this would be less efficient.It's important to use the Java Wrapper (e.g. java.lang.Long) for which Hibernate can check for nullability, instead of a primitive (e.g. long) for the @Version property. The same rules apply to the Spring Data save method as well. If you ever use an assigned identifier generator, you have to remember to add a Java Wrapper @Version property, otherwise, a redundant SELECT statement is going to be generated.

---

97. By now, it's clear that new entities must go through persist, whereas detached entities must be reattached using merge. However, while reviewing lots of projects, I came to realize that the following anti-pattern is rather widespread:

```
@Transactional
public void savePostTitle(Long postId, String title) {
  Post post = postRepository.findOne(postId);
  post.setTitle(title);
  postRepository.save(post);
}
```

The save method serves no purpose. Even if we remove it, Hibernate will still issue the UPDATE statement since the entity is managed and any state change is propagated as long as the currently running EntityManager is open. This is an anti-pattern because the save call fires a MergeEvent which is handled by the DefaultMergeEventListener.

---

98. While a save() method might be convenient in some situations, in practice, you should never call

merge() for entities that are either new or already managed. As a rule of thumb, you shouldn't be using save() with JPA. For new entities, you should always use persist(), while for detached entities you need to call merge(). For managed entities, you don't need any save() method because Hibernate automatically synchronizes the entity state with the underlying database record.

---

99. Not all queries trigger a Session flush

Many would assume that Hibernate always flushes the Session before any executing query. While this might have been a more intuitive approach, and probably closer to the JPA's AUTO FlushModeType, Hibernate tries to optimize that. If the current executed query is not going to hit the pending SQL INSERT/UPDATE/DELETE statements then the flush is not strictly required.

As stated in the reference documentation, the AUTO flush strategy may sometimes synchronize the current persistence context prior to a query execution. It would have been more intuitive if the framework authors had chosen to name it FlushMode.SOMETIMES.

---

100. AUTO flush and native SQL queries

```
Product product = new Product();
session.persist(product);
assertNull(session.createSQLQuery("select id from product").uniqueResult());
```

```
DEBUG [main]: o.h.e.i.AbstractSaveEventListener -
        Generated identifier: 718b84d8-9270-48f3-86ff-0b8da7f9af7c,
                                using strategy: org.hibernate.id.UUIDGenerator
Query:{[select id from product][]}
Query:{[insert into product (color, id) values (?, ?)][12,718b84d8-9270-48f3-86ff-0b8da7f9af7c]}
```

The newly persisted Product was only inserted during transaction commit, because the native SQL query didn't triggered the flush. This is major consistency problem, one that's hard to debug or even foreseen by many developers. That's one more reason for always inspecting auto-generated SQL statements.

---

101. The same behaviour is observed even for named native queries:

```
@NamedNativeQueries(
@NamedNativeQuery(name = "product_ids", query = "select id from product")
)
assertNull(session.getNamedQuery("product_ids").uniqueResult());
```

So even if the SQL query is pre-loaded, Hibernate won't extract the associated query space for matching it against the pending DML statements.

---

102. Query flush mode

The ALWAYS mode is going to flush the persistence context before any query execution (HQL or SQL). This time, Hibernate applies no optimization and all pending entity state transitions are going to be synchronized with the current database transaction.

103.
```
assertEquals(product.getId(), session.createSQLQuery("select id from product").
        setFlushMode(FlushMode.ALWAYS).uniqueResult());
```

---

104. Instructing Hibernate which tables should be synchronized

You could also add a synchronization rule on your current executing SQL query. Hibernate will then know what database tables need to be synchronized prior to executing the query. This is also useful for second level caching as well.

```
assertEquals(product.getId(), session.createSQLQuery("select id from product").
        addSynchronizedEntityClass(Product.class).uniqueResult());
```

---

105. The AUTO flush mode is tricky and fixing consistency issues on a query basis is a maintainer's nightmare. If you decide to add a database trigger, you'll have to check all Hibernate queries to make sure they won't end up running against stale data.

My suggestion is to use the ALWAYS flush mode, even if Hibernate authors warned us that: this strategy is almost always unnecessary and inefficient.

---

106. Inconsistency is much more of an issue that some occasional premature flushes. While mixing DML operations and queries may cause unnecessary flushing this situation is not that difficult to mitigate. During a session transaction, it's best to execute queries at the beginning (when

no pending entity state transitions are to be synchronized) and towards the end of the transaction (when the current persistence context is going to be flushed anyway).

The entity state transition operations should be pushed towards the end of the transaction, trying to avoid interleaving them with query operations (therefore preventing a premature flush trigger).

---

107. When using the default AUTO flush mode, the Persistence Context should be flushed as follows:

- before the transaction is committed,
- before running a JPQL or HQL query
- before executing a native SQL query

As previously explained, Hibernate triggers the AUTO flush only for the first two events, and the native SQL queries must either override the ALWAYS flush mode using the Query#setFlushMode(FlushMode flushMode) method or add a table space synchronization (e.g. SQLQuery#addSynchronizedEntityClass(Class entityClass), SQLQuery#addSynchronizedEntityName(String entityName), SQLQuery#addSynchronizedQuerySpace(String querySpace)).

This is only required for the native Hibernate API, when using a `Session` explicitly.

JPA is more strict, and the AUTO flush mode must trigger a flush before any query. More the section 3.10.8 of the Java Persistence API specification says that the AUTO flush mode should ensure that all pending changes are visible by any executing query.

---

108. According to Hibernate Docs entity may be in one of the following states:

- new/transient: the entity is not associated to a persistence context, be it a newly created object the database doesn't know anything about.
- persistent: the entity is associated to a persistence context (residing in the 1st Level Cache) and there is a database row representing this entity.
- detached: the entity was previously associated to a persistence context, but the persistence context was closed, or the entity was manually evicted.
- removed: the entity was marked as removed and the persistence context will remove it from the database at flush time.

---

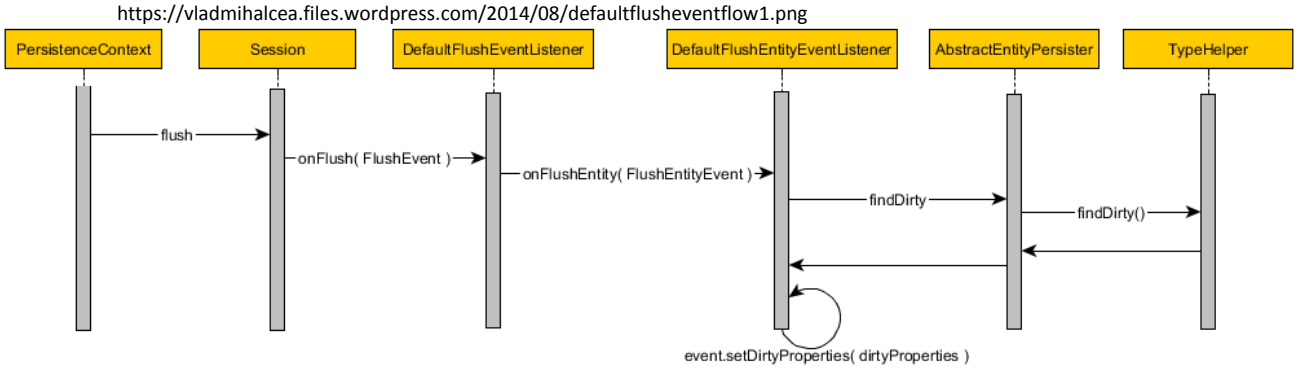
109. According to Hibernate JavaDocs the SQL operations order is:

inserts
updates
deletions of collections elements
inserts of the collection elements
deletes

---

110. The persistence context enqueues entity state transitions that get translated to database statements upon flushing. For managed entities, Hibernate can auto-detect incoming changes and schedule SQL UPDATES on our behalf. This mechanism is called automatic dirty checking.

https://vladmihalcea.files.wordpress.com/2014/08/defaultflusheventflow1.png



---

111. Even if only one property of a single entity has ever changed, Hibernate will still check all managed entities. For a large number of managed entities, the default dirty checking mechanism may have a significant CPU and memory footprint. Since the initial entity snapshot is held separately, the persistence context requires twice as much memory as all managed entities would normally occupy.

---

112. To instrument all @Entity classes, you need to add the following Maven plugin (in Hibernate 5):

```
<plugin>
    <groupId>org.hibernate.orm.tooling</groupId>
    <artifactId>hibernate-enhance-maven-plugin</artifactId>
    <version>${hibernate.version}</version>
    <executions>
        <execution>
            <configuration>
                <enableDirtyTracking>true</enableDirtyTracking>
            </configuration>
            <goals>
                <goal>enhance</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

After the Java classes are compiled, the plugin goes through all entity classes and modifies their bytecode according to the instrumentation options chosen during configuration.

---

113. Although bytecode enhancement dirty tracking can speed up the Persistence Context flushing mechanism, if the size of the Persistence Context is rather small, the improvement is not that significant.
The entity snapshot is still saved in the Persistence Context even when using bytecode enhancement.
For this reason, keeping the Persistence Context in reasonable boundaries holds on no matter the dirty tracking mechanism in use.

---

114. The JDBC ResultSet offers a client-side Proxy cursor for fetching the current statement return data.
When the statement gets executed, the result must be transferred from the database cursor to the client-side one. This operation can either be done at once or on demand.

---

115. TYPE_FORWARD_ONLY
This is the default ResultSet cursor type. The result set can only be moved forward and the resulted data can either be fetched at once or retrieved while the cursor is being iterated.
The database can decide to fetch the data as it was available at the time the query started or as it is upon fetching.

TYPE_SCROLL_INSENSITIVE
The result set can be scrolled both forward and backward and the resulted data is insensitive to concurrent changes occurring while the cursor is still open

TYPE_SCROLL_SENSITIVE
The result set can be scrolled both forward and backward and the resulted data is sensitive to concurrent changes occurring while the cursor is still open. The data is therefore fetched on demand as opposed to being retrieved from a database cursor cache

---

116. The Java Persistence Query interface offers only full-result retrievals, through the Query.getResultList() method call.

Hibernate also supports scrollable ResultSet cursors through its specific Query.scroll() API.

The only apparent advantage of scrollable ResultSets is that we can avoid memory issues on the client-side, since data is being fetched on demand. This might sound like a natural choice, but in reality, you shouldn't fetch large result sets for the following reasons:

- Large result sets impose significant database server resources and because a database is a highly concurrent environment, it might hinder availability and scalability

- Tables tend to grow in size and a moderate result set might easily turn into a very large one. This kind of situation happens in production systems, long after the application code was shipped. Because users can only browse a relatively small portion of the whole result set, pagination is a more scalable data fetching alternative

- The overly common offset paging is not suitable for large result sets (because the response time increases linearly with the page number) and you should consider keyset pagination when traversing large result sets. The keyset pagination offers a constant response time insensitive to the relative position of the page being fetched

- Even for batch processing jobs, it's always safer to restrict processing items to a moderate batch size. Large batches can lead to memory issues or cause long-running transactions, which increase the undo/redo transaction log size

---

117. Did you know pagination with offset is very troublesome but easy to avoid?

Offset instructs the databases skip the first N results of a query. However, the database must still fetch these rows from the disk and bring them in order before it can send the following ones.

In other words, big offsets impose a lot of work on the database—no matter whether SQL or NoSQL.

But the trouble with offset doesn't stop here: think about what happens if a new row is inserted between fetching two pages?

---

118. Life Without OFFSET
Now imagine a world without these problems. As it turns out, living without offset is quite simple: just use a where clause that selects only data you haven't seen yet.

For that, we exploit the fact that we work on an ordered set—you do have an order by clause, ain't you? Once there is a definite sort order, we can use a simple filter to only select what follows the entry we have see last:

```
SELECT ...
  FROM ...
       WHERE ...
          AND id < ?last_seen_id
               ORDER BY id DESC
                FETCH FIRST 10 ROWS ONLY
```

This is the basic recipe. It gets more interesting when sorting on multiple columns, but the idea is the same. This recipe is also applicable to many NoSQL systems.

This approach—called seek method or keyset pagination—solves the problem of drifting results as illustrated above and is even faster than offset. If you'd like to know what happens inside the database when using offset or keyset pagination, have a look at these slides (benchmarks, benchmarks!):

---

119. To configure Hibernate to use an explicit Statement fetchSize, we need to set the following Hibernate property:

```
properties.put("hibernate.jdbc.fetch_size", fetchSize());
```

---

120. The default fetch size yields the best result, even when the fetchSize is equal to the total number of rows being returned. Since there is no upper-bound buffer limit, the default fetch size can cause OutOfMemoryError issues when retrieving large result sets.

---

121. While most database serves don't impose a default upper limit for the result set fetch size, it's a good practice to limit the whole result set (if requirements allow it). A limited size result set should address the unbounded fetch size shortcoming, while ensuring predictable response times even when the queried data grows gradually. The shorter the queries, the quicker the row-level locks are released and the more scalable the data access layer becomes.

---

122. When JPA loads an entity it also loads all the EAGER or "join fetch" associations too. As long as the persistence context is opened, navigating the LAZY associations results in fetching those as well, through additional executed queries.

By default, the JPA @ManyToOne and @OneToOne annotations are fetched EAGERly, while the @OneToMany and @ManyToMany relationships are considered LAZY. This is the default strategy, and Hibernate doesn't magically optimizes your object retrieval, it only does what is instructed to do.

---

123. The default fetch strategy is the one you define through the JPA mapping, while the manual join fetching is when you use JPQL queries.

The best advice I can give you is to favour the manual fetching strategy (defined in JPQL queries using the fetch operator). While some @ManyToOne or @OneToOne associations make sense to always be fetched eagerly, most of the time, they aren't needed for every fetching operation.

For children associations it's always safer to mark them LAZY and only "join fetch" them when needed, because those can easily generate large SQL result sets, with unneeded joins.

---

124. Having most of the associations defined as LAZY requires us to use the "join fetch" JPQL operator and retrieve only the associations we need to fulfil a given request. If you forget to "join fetch" properly, the Persistence Context will run queries on your behalf while you navigate the lazy associations, and that might generate "N+1" problems, or additional SQL queries which might have

been retrieved with a simple join in the first place.

---

125. It's a good practice to set the default fetch strategy explicitly (it makes the code more self-descriptive) even if @ManyToOne uses the EAGER fetch option by default. Every time we load through the entity manager the default fetching strategy comes into play. The JPQL and Criteria queries might override the default fetch plan.

---

126. For LAZY associations, all uninitialized proxies are vulnerable to LazyInitializationException, if accessed from within a closed Persistence Context. If the Persistence Context is still open it will generate additional select queries, which might end up in N+1 query issues.

---

127. When our children associations are mapped as Bags, the same JPQL query throws a org.hibernate.loader.MultipleBagFetchException. This is not happening for Sets or Indexed Lists.

---

128. Hibernate defines four association retrieving strategies:

Join	The association is OUTER JOINED in the original SELECT statement
Select	An additional SELECT statement is used to retrieve the associated entity(entities)
Subselect	An additional SELECT statement is used to retrieve the whole associated collection. This mode is meant for to-many associations.
Batch	An additional number of SELECT statements is used to retrieve the whole associated collection. Each additional SELECT will retrieve a fixed number of associated entities. This mode is meant for to-many associations.

These fetching strategies might be applied in the following scenarios:

- the association is always initialized along with its owner (e.g. EAGER FetchType)
- the uninitialized association (e.g. LAZY FetchType) is navigated, therefore the association must be retrieved with a secondary SELECT.

The Hibernate mappings fetching information forms the global fetch plan. At query time, we may override the global fetch plan, but only for LAZY associations. For this we can use the fetch HQL/JPQL/Criteria directive. EAGER associations cannot be overridden, therefore tying your application to the global fetch plan.

---

129. JPQL example:

```
Product product = entityManager.createQuery("select p " + "from Product p " +
                  "where p.id = :productId", Product.class)
                 .setParameter("productId", productId)
                 .getSingleResult();
```

CriteriaBuilder alternative:

```
CriteriaBuilder cb = entityManager.getCriteriaBuilder();
CriteriaQuery<Product> cq = cb.createQuery(Product.class);
Root<Product> productRoot = cq.from(Product.class);
cq.where(cb.equal(productRoot.get("id"), productId));
Product product = entityManager.createQuery(cq).getSingleResult();
```

---

130. An index is a distinct structure in the database that is built using the create index statement. It requires its own disk space and holds a copy of the indexed table data. That means that an index is pure redundancy. A database index is, after all, very much like the index at the end of a book: it occupies its own space, it is highly redundant, and it refers to the actual information stored in a different place.

---

131. Transactions are concurrency control mechanisms, and they deliver consistency even when being interleaved. Isolation brings us the benefit of hiding uncommitted state changes from the outside world, as failing transactions shouldn't ever corrupt the state of the system. Isolation is achieved through concurrency control using pessimistic or optimistic locking mechanisms.

---

132. The SQL standard defines four Isolation levels:

READ_UNCOMMITTED
READ_COMMITTED
REPEATABLE_READ
SERIALIZABLE

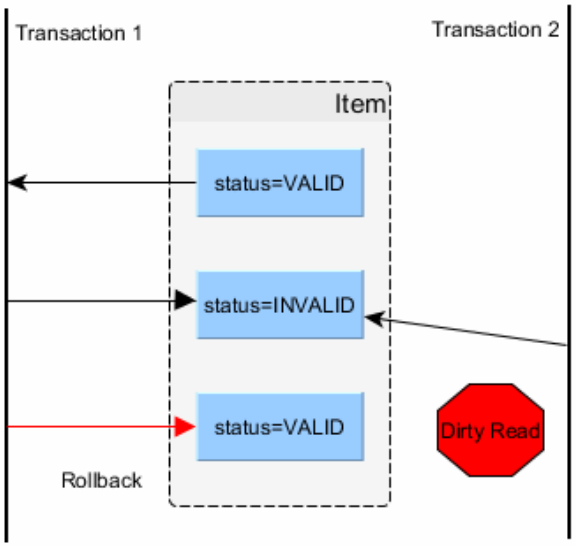
---

133. All but the SERIALIZABLE level are subject to data anomalies (phenomena) that might occur according to the following pattern:

Isolation Level	Dirty read	Non-repeatable read	Phantom read
READ_UNCOMMITTED	allowed	allowed	allowed
READ_COMMITTED	prevented	allowed	allowed
REPEATABLE_READ	prevented	prevented	allowed
SERIALIZABLE	prevented	prevented	prevented

---

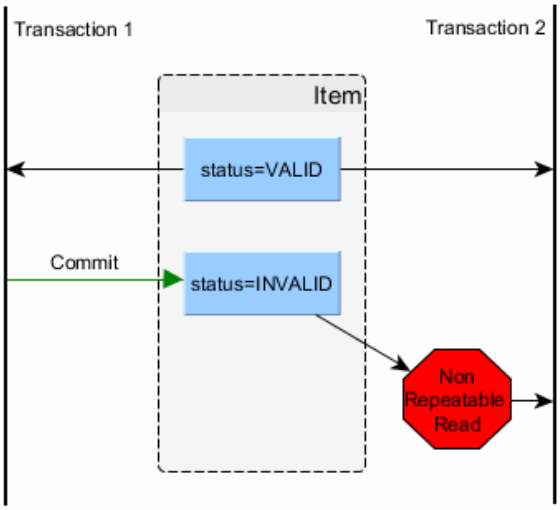
134. Dirty read phenomena
(https://vladmihalcea.files.wordpress.com/2014/01/acid-dirty-read.gif)



A dirty read happens when a transaction is allowed to read uncommitted changes of some other running transaction. This happens because there is no locking preventing it. In the picture above, you can see that the second transaction uses an inconsistent value as of the first transaction had rollbacked.

---

135. Non-repeatable read phenomena
(https://vladmihalcea.files.wordpress.com/2014/01/acid-non-repeatable-read.gif)

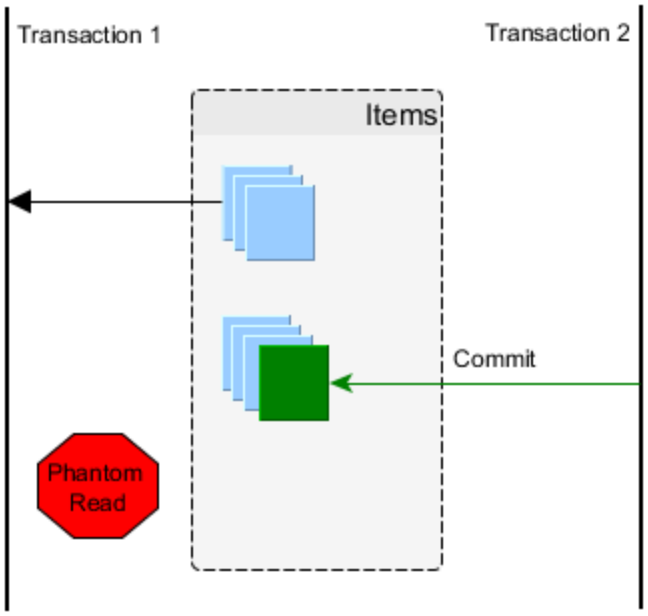


A non-repeatable read manifests when consecutive reads yield different results due to a concurring transaction that has just updated the record we're reading. This is undesirable since we end up using stale (statut) data. This is prevented by holding a shared lock (read lock) on the read record for the whole duration of the current transaction.

---

136. A phantom read happens when a second transaction inserts a row that matches a previous select criteria of the first transaction. We, therefore, end up using stale data, which might affect our business operation. This is prevented using range locks or predicate locking.

https://vladmihalcea.com/2014/01/05/a-beginners-guide-to-acid-and-database-transactions/



---

137. Usually, READ_COMMITED is the right choice, since not even SERIALIZABLE can protect you from a lost update where the reads/writes happen in different transactions (and web requests). You should take into consideration your enterprise system requirements and set up tests for deciding which isolation level best suits your needs.

---

138. An logical transaction is an application-level unit of work that may span over multiple physical (database) transactions. Holding the database connection open throughout several user requests, i including user think time, is definitely an anti-pattern.

---

139. Hibernate offers two strategies for implementing long conversations:

Extended persistence context
Detached objects

---

140. For disabling persistence in the course of the application-level transaction, we have the following options:

We can disable automatic flushing, by switching the Session FlushMode to MANUAL. At the end of the last physical transaction, we need to explicitly call Session#flush() to propagate the entity state transitions.
All but the last transaction are marked read-only. For read-only transactions Hibernate disables both dirty checking and the default automatic flushing.
The read-only flag might propagate to the underlying JDBC Connection, so the driver might enable some database-level read-only optimizations.

The last transaction must be writeable so that all changes are flushed and committed.

---

141. Optimistic locking works for both database and application-level transactions, and it doesn't make use of any additional database locking. Optimistic locking can prevent lost updates and that's why I always recommend all entities be annotated with the @Version attribute.

---

142. Application-level transactions offer a suitable concurrency control mechanism for long conversations. All entities are loaded within the context of a Hibernate Session, acting as a transactional write-behind cache.

A Hibernate persistence context can hold one and only one reference of a given entity.
The first level cache guarantees session-level repeatable reads.

If the conversation spans over multiple requests we can have application-level repeatable reads. Long conversations are inherently stateful so we can opt for detached objects or long persistence contexts. But application-level repeatable reads require an application-level concurrency control strategy such as optimistic locking.

---

143. Write-behind

Hibernate tries to defer the Persistence Context flushing up until the last possible moment.

This strategy has been traditionally known as transactional write-behind.

The write-behind is more related to Hibernate flushing rather than any logical or physical transaction. During a transaction, the flush may occur multiple times.

The flushed changes are visible only for the current database transaction. Until the current transaction is committed, no change is visible by other concurrent transactions.

The persistence context, also known as the first level cache, acts as a buffer between the current entity state transitions and the database.

In caching theory, the write-behind synchronization requires that all changes happen against the cache, whose responsibility is to eventually synchronize with the backing store.

144.

READ UNCOMMITTED
READ COMMITTED (protecting against dirty reads)
REPEATABLE READ (protecting against dirty and non-repeatable reads)
SERIALIZABLE (protecting against dirty, non-repeatable reads and phantom reads)

Although the most consistent SERIALIZABLE isolation level would be the safest choice, most databases default to READ COMMITTED instead. Apart from MySQL (which uses REPEATABLE_READ), the default isolation level of most relational database systems is READ_COMMITTED. All databases allow you to set the default transaction isolation level.

145.

Read committed is an isolation level that guarantees that any data read was committed at the moment is read. It simply restricts the reader from seeing any intermediate, uncommitted, 'dirty' read. IT makes no promise whatsoever that if the transaction re-issues the read, will find the Same data, data is free to change after it was read.

Repeatable read is a higher isolation level, that in addition to the guarantees of the read committed level, it also guarantees that any data read cannot change, if the transaction reads the same data again, it will find the previously read data in place, unchanged, and available to read.

The next isolation level, serializable, makes an even stronger guarantee: in addition to everything repeatable read guarantees, it also guarantees that no new data can be seen by a subsequent read.

Two selects in the same transaction with a pause of 1 minute:

```
BEGIN TRANSACTION;
        SELECT * FROM T;
        WAITFOR DELAY '00:01:00'
        SELECT * FROM T;
COMMIT;
```

- READ COMITTED (protecting against dirty reads): the second SELECT may return any data. A concurrent transaction may update the record, delete it, insert new records. The second select will always see the new data.

- REPEATABLE READ: The second SELECT is guaranteed to see the rows that has seen at first select unchanged.This is the default isolation level for InnoDB. For consistent reads, there is an important difference from the READ COMMITTED isolation level: All consistent reads within the same transaction read the snapshot established by the first read. This convention means that if you issue several plain (nonlocking) SELECT statements within the same transaction, these SELECT statements are consistent also with respect to each other.

- SERIALIZABLE reads the second select is guaranteed to see exactly the same rows as the first. No row can change, nor deleted, nor new rows could be inserted by a concurrent transaction.

READ COMMITTED (default)
Shared locks are taken in the SELECT and then released when the SELECT statement completes. This is how the system can guarantee that there are no dirty reads of uncommitted data. Other transactions can still change the underlying rows after your SELECT completes and before your transaction completes.

REPEATABLE READ
Shared locks are taken in the SELECT and then released only after the transaction completes. This is how the system can guarantee that the values you read will not change during the transaction (because they remain locked until the transaction finishes). This level is different from Read Committed in that a query in a repeatable read transaction sees a snapshot as of the start of the transaction, not as of the start of the current query within the transaction.

There are two processes A and B. Process B is reading Table X Process A is writing in table X Process B is reading again Table X.

ReadUncommitted: Process B can read uncommitted data from process A and it could see different rows based on B writing. No lock at all
ReadCommitted: Process B can read ONLY committed data from process A and it could see different rows based on COMMITTED only B writing. could we call it Simple Lock?
RepeatableRead: Process B will read the same data (rows) whatever Process A is doing. But process A can change other rows. Rows level Block
Serialisable: Process B will read the same rows as before and Process A cannot read or write in the table. Table-level Block
Snapshot: every process has its own copy and they are working on it. Each one has its own view

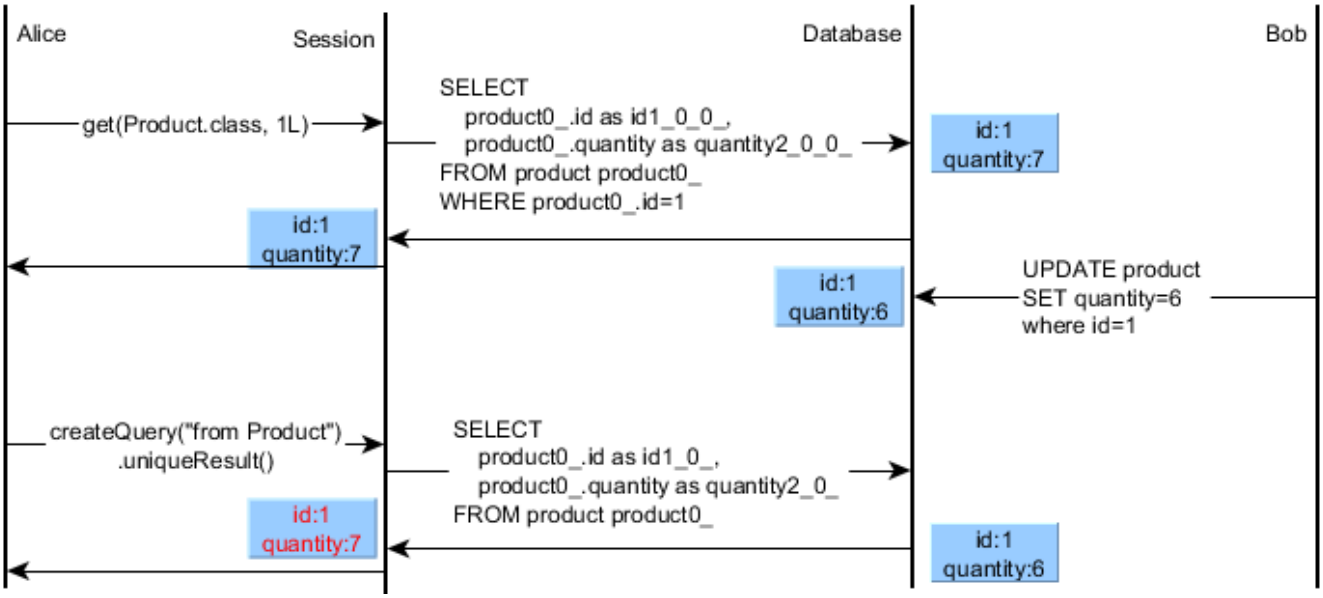
---

146. Dirty read: An operation that retrieves unreliable data, data that was updated by another transaction but not yet committed. It is only possible with the isolation level known as read uncommitted.

---

147. In repeatable read, if your Hibernate Session has already loaded a given entity then any successive entity query (JPQL/HQL) is going to return the very same object reference (disregarding the current loaded database snapshot):

https://vladmihalcea.files.wordpress.com/2014/10/applicationlevelrepeatableread.png



The differences between entity queries and SQL projections. While SQL query projections always load the latest database state, entity query results are managed by the first level cache, ensuring session-level repeatable reads.

Workaround 1: If your use case demands reloading the latest database entity state then you can simply use the Session#refresh() to refresh the entity in question.
Workaround 2: If you want an entity to be disassociated from the Hibernate first level cache you can easily evict it (Session#evict()), so the next entity query can use the latest database entity value.

---

148. While native SQL remains the de facto relational data reading technique, Hibernate excels in writing data. Hibernate is a persistence framework and you should never forget that. Loading entities make sense if you plan on propagating changes back to the database. You don't need to load entities for displaying read-only views, an SQL projection being a much better alternative in this case.

---

149. Session-level repeatable reads prevent lost updates in concurrent writes scenarios, so there's a good reason why entities don't get refreshed automatically. Maybe we've chosen to manually flush dirty properties and an automated entity refresh might overwrite synchronized pending changes.

---

150. In relational databases, two records are associated through a foreign key reference. In this relationship, the referenced record is the parent while the referencing row (the foreign key side) is the child. A non-null foreign key may only reference an existing parent record.

In the Object-oriented space this association can be represented in both directions. We can have a many-to-one reference from a child to parent and the parent can also have a one-to-many children collection.

Because both sides could potentially control the database foreign key state, we must ensure that only one side is the owner of this association. Only the owning side state changes are propagated to the database. The non-owning side has been traditionally referred as the inverse side

---

151. The unidirectional parent-owning-side-child association mapping
Only the parent side has a @OneToMany non-inverse children collection. The child entity doesn't reference the parent entity at all.

```
@Entity(name = "post")
public class Post {
    ...
    @OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)
    private List<Comment> comments = new ArrayList<Comment>();
    ...
}
```

---

152. The unidirectional parent-owning-side-child component association mapping
The child side doesn't always have to be an entity and we might model it as a component type instead. An Embeddable object (component type) may contain both basic types and association mappings but it can never contain an @Id. The Embeddable object is persisted/removed along with its owning entity.

The parent has an @ElementCollection children association. The child entity may only reference the parent through the non-queryable Hibernate specific @Parent annotation.

```
@Entity(name = "post")
public class Post {
    ...
    @ElementCollection
    @JoinTable(name = "post_comments", joinColumns = @JoinColumn(name = "post_id"))
    @OrderColumn(name = "comment_index")
    private List<Comment> comments = new ArrayList<Comment>();
    ...

    public void addComment(Comment comment) {
        comment.setPost(this);
        comments.add(comment);
    }
}

@Embeddable
public class Comment {
    ...
    @Parent
    private Post post;
    ...
}
```

---

153. The bidirectional parent-owning-side-child association mapping
The parent is the owning side so it has a @OneToMany non-inverse (without a mappedBy directive) children collection. The child entity references the parent entity through a @ManyToOne association that's neither insertable nor updatable:

```
@Entity(name = "post")
public class Post {
    ...
    @OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)
    private List<Comment> comments = new ArrayList<Comment>();
    ...

    public void addComment(Comment comment) {
        comment.setPost(this);
        comments.add(comment);
    }
}

@Entity(name = "comment")
public class Comment {
    ...
    @ManyToOne
    @JoinColumn(name = "post_id", insertable = false, updatable = false)
    private Post post;
    ...
```

```
}
```

---

154. The bidirectional child-owning-side-parent association mapping

The child entity references the parent entity through a @ManyToOne association, and the parent has a mappedBy @OneToMany children collection. The parent side is the inverse side so only the @ManyToOne state changes are propagated to the database.

Even if there's only one owning side, it's always a good practice to keep both sides in sync by using the add/removeChild() methods.

```
@Entity(name = "post")
public class Post {
    ...
    @OneToMany(cascade = CascadeType.ALL, orphanRemoval = true, mappedBy = "post")
    private List<Comment> comments = new ArrayList<Comment>();
    ...

    public void addComment(Comment comment) {
        comment.setPost(this);
        comments.add(comment);
    }
}

@Entity(name = "comment")
public class Comment {
    ...
    @ManyToOne
    private Post post;
    ...
}
```

---

155. The unidirectional child-owning-side-parent association mapping

The child entity references the parent through a @ManyToOne association. The parent doesn't have a @OneToMany children collection so the child entity becomes the owning side. This association mapping resembles the relational data foreign key linkage.

```
@Entity(name = "comment")
public class Comment {
    ...
    @ManyToOne
    private Post post;
    ...
}
```

---

156. Overruling default collection versioning

If the default owning-side collection versioning is not suitable for your use case, you can always overrule it with Hibernate @OptimisticLock annotation. Let's overrule the default parent version update mechanism for bidirectional parent-owning-side-child association:

```
@Entity(name = "post")
public class Post {
    ...
    @OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)
    @OptimisticLock(excluded = true)
    private List<Comment> comments = new ArrayList<Comment>();
    ...

    public void addComment(Comment comment) {
        comment.setPost(this);
        comments.add(comment);
    }
}

@Entity(name = "comment")
public class Comment {
    ...
    @ManyToOne
    @JoinColumn(name = "post_id", insertable = false, updatable = false)
    private Post post;
    ...
}
```

It's very important to understand how various modelling structures impact concurrency patterns. The owning-side collections changes are taken into consideration when incrementing the parent version number, and you can always bypass it using the @OptimisticLock annotation.

---

157. Enabling entity-level optimistic locking is fairly easy. You just have to mark one logical-clock property (usually an integer counter) with the JPA @Version annotation and Hibernate takes care of the rest.

---

158. Version-less optimistic locking

Optimistic locking is commonly associated with a logical or physical clocking sequence, for both performance and consistency reasons. The clocking sequence points to an absolute entity state version for all entity state transitions.

To support legacy database schema optimistic locking, Hibernate added a version-less concurrency control mechanism. To enable this feature you have to configure your entities with the @OptimisticLocking annotation that takes the following parameters:

Optimistic Locking Type	Description
ALL	All entity properties are going to be used to verify the entity version
DIRTY	Only current dirty properties are going to be used to verify the entity version
NONE	Disables optimistic locking
VERSION	Surrogate version column optimistic locking

For version-less optimistic locking, you need to choose ALL or DIRTY.

---

159. First thing to notice is the absence of a surrogate version column. For concurrency control, we'll use DIRTY properties optimistic locking:

```
@Entity(name = "product")
@Table(name = "product")
@OptimisticLocking(type = OptimisticLockType.DIRTY)
@DynamicUpdate
public class Product {
 //code omitted for brevity
}
```

By default, Hibernate includes all table columns in every entity update, therefore reusing cached prepared statements. For dirty properties optimistic locking, the changed columns are included in the update WHERE clause and that's the reason for using the @DynamicUpdate annotation.

---

160. The version-less optimistic locking is a viable alternative as long as you can stick to a non-detached entities policy. Combined with extended persistence contexts, this strategy can boost writing performance even for a legacy database schema.

---

161. Merging

The merge operation consists in loading and attaching a new entity object from the database and update it with the currently given entity snapshot. Merging is supported by JPA too and it's tolerant to already managed Persistence Context entity entries. If there's an already managed entity then the select is not going to be issued, as Hibernate guarantees session-level repeatable reads.

---

162. Reattaching

Reattaching is a Hibernate specific operation. As opposed to merging, the given detached entity must become managed in another Session. If there's an already loaded entity, Hibernate will throw an exception. This operation also requires an SQL SELECT for loading the current database entity snapshot. The detached entity state will be copied on the freshly loaded entity snapshot and the dirty checking mechanism will trigger the actual DML update.

---

163. Resource local transactions are used in JSE, or in application managed (non-managed) mode in Java EE. To use resource local transactions the transaction-type attribute in the persistence.xml is set to RESOURCE_LOCAL. If resource local transactions are used with a DataSource, the <non-jta-data-source> element should be used to reference a server DataSource that has been configure to not be JTA managed.

Local JPA transactions are defined through the EntityTransaction class. It contains basic transaction API including begin, commit and rollback.

Technically in JPA the EntityManager is in a transaction from the point it is created. So begin is somewhat redundant. Until begin is called, certain operations such as persist, merge, remove cannot be called. Queries can still be performed, and objects that were queried can be changed, although this is somewhat unspecified what will happen to these changes in the JPA spec, normally they will be committed, however it is best to call begin before making any changes to your objects. Normally it is best to create a new EntityManager for each transaction to avoid have stale objects remaining in the persistence context, and to allow previously managed objects to garbage collect.

After a successful commit the EntityManager can continue to be used, and all of the managed objects remain managed. However it is normally best to close or clear the EntityManager to allow garbage collection and avoid stale data. If the commit fails, then the managed objects are considered detached, and the EntityManager is cleared. This means that commit failures cannot be caught and retried, if a failure occurs, the entire transaction must be performed again. The previously managed object may also be left in an inconsistent state, meaning some of the objects locking version may have been incremented. Commit will also fail if the transaction has been marked for rollback. This can occur either explicitly by calling setRollbackOnly or is required to be set if any query or find operation fails. This can be an issue, as some queries may fail, but may not be desired to cause the entire transaction to be rolled back.

The rollback operation will rollback the database transaction only. The managed objects in the persistence context will become detached and the EntityManager is cleared. This means any object previously read, should no longer be used, and is no longer part of the persistence context. The changes made to the objects will be left as is, the object changes will not be reverted.

---

164. JTA transactions are used in Java EE, in managed mode (EJB). To use JTA transactions the transaction-type attribute in the persistence.xml is set to JTA. If JTA transactions are used with a DataSource, the <jta-datasource> element should be used to reference a server DataSource that has been configure to be JTA managed.

JTA transactions are defined through the JTA UserTransaction class, or more likely implicitly defined through SessionBean usage/methods. In a SessionBean each SessionBean method defaults to a TransactionAttribute TransactionAttributeType.REQUIRED and hence it's invocation starts a JTA transaction, if no JTA trasnsaction is in progress. If a method is the first called that requires a transaction, it will commit the changes upon completion.

JTA transaction can be are shared among SessionBean methods, hence if a method is called through it's business interface from another method that already started a transaction, the second method will work inside the existing transaction. Moreover the second method can only change the state of the EntityManager by persisting objects or doing other operations, and when it returns control to the calling method it delegates to it the execution of the actual commit. To perform DB operations in a transaction isolated from any transaction in progress a SessionBean method can be annotated as TransactionAttribute TransactionAttributeType.REQUIRES_NEW. The TransactionAttributeTypes that can be associate to SessionBean methods are:

REQUIRED	Requires a transaction: if the client invoking the method is already associated with a transactional context, the method is hosted in the client's context, otherwise a new transaction is started
REQUIRES_NEW	Requires a separate transaction: starts a transaction in any case, if the method is called by a client already associated with a transactional context, the existing transaction will be suspended
MANDATORY	Requires an existing transaction: it requires that the client invoking the method is already associated with a transactional context, and the method is hosted in the client's context
NOT_SUPPORTED	Does not use any transaction: no transaction is started, if the client invoking the method is already associated with a transactional context, the existing transaction will be suspended
NEVER	Cannot execute if a transaction is in place: the client is required to call outside of any transactional context
SUPPORTS	Does not require a transaction, but if the client invoking the method is already associated with a transactional context the method will execute in the transaction context

---

165. Delegating the whole locking responsibility to the database system can both simplify application development and prevent concurrency issues, such as deadlocking. Deadlocks can still occur, but the database can detect and take safety measures (arbitrarily releasing one of the two competing locks).

---

166. XA was only required for multiple data sources (or other branches in the transaction e.g JMS).

---

167. JTA/XA is a kind of system insurance against data corruption (and the resulting business losses).
The most common use cases are:

- Processing JMS messages from a queue and inserting the results in a database: you don't want a crash to lose messages whose results are not yet stored in the database.
- Updating two or more legacy back-end systems in the same transaction

In general, whenever you access more than one back-end system in the same transaction the use of JTA/XA is highly recommended. Otherwise, the risk of data loss or corruption is too high (and not necessarily visible!).

A sample calculation shows the expected benefit of JTA/XA: suppose you are a bank and you have to handle 1 million transactions per day, and with an average value per transaction of 100 USD. This makes a total turnover of 100 million USD per day. If a system crash would cause as little as 1% data loss, then you would have a total cost of 1 million USD for a crash! In this scenario, JTA/XA clearly saves you a lot of money.

With other words:
In computing, the XA standard is a specification by The Open Group for distributed transaction processing (DTP). It describes the interface between the global transaction manager and the local resource manager. The goal of XA is to allow multiple resources (such as databases, application servers, message queues, transactional caches, etc.) to be accessed within the same transaction, thereby preserving the ACID properties across applications. XA uses a two-phase commit to ensure that all resources either commit or roll back any particular transaction consistently (all do the same).

XA stands for "eXtended Architecture" and is an X/Open group standard for executing a "global transaction" that accesses more than one back-end data-store. XA specifies how a transaction manager will roll up the transactions against the different data-stores into an "atomic" transaction and execute this with the two-phase commit (2PC) protocol for the transaction. Thus, XA is a type of transaction coordination, often among databases. ACID Transactions are a key feature of databases, but typically databases only provide the ACID guarantees for activities that happen inside a single database. XA coordination allows many resources (again, often databases) to participate in a single, coordinated, atomic update operation.

The XA specification describes what a resource manager must do to support transactional access. Resource managers that follow this specification are said to be XA-compliant.

The XA specification was based on an interface used in the Tuxedo system developed in the 1980s

---

168. The Java Transaction API (JTA) is one of the Java Enterprise Edition (Java EE) APIs allowing distributed transactions to be done across multiple XA resources in a Java environment.

---

169. JTA is a general API for managing transactions in Java. It allows you to start, commit and rollback transactions in a resource neutral way. Transactional status is typically stored in TLS (Thread Local Storage) and can be propagated to other methods in a call-stack without needing some explicit context object to be passed around. Transactional resources can join the ongoing transaction. If there is more than one resource participating in such a transaction, at least one of them has to be a so-called XA resource.

A resource local transaction is a transaction that you have with a specific single resource using its own specific API. Such a transaction typically does not propagate to other methods in a call-stack and you are required to pass some explicit context object around. In the majority of the resource local transactions it's not possible to have multiple resources participating in the same transaction.

You would use a resource local transaction in for instance low-level JDBC code in Java SE.
Here the context object is expressed by an instance of java.sql.Connection. Other examples of resource local transactions are developers creating enterprise applications around 2002. Since transaction managers (used by JTA) were expensive, closed source and complicated things to setup around that era, people went with the cheaper and easier to obtain resource local variants.

You would use a JTA transaction in basically every other scenario. Very simple, small, free and o pen-source servers like TomEE (25MB) or GlassFish (35MB) have JTA support out of the box. There's nothing to setup and they Just Work.

Finally, technologies like EJB and Spring make even JTA easier to use by offering declarative

transactions. In most cases it's advised to use those as they are easier, cleaner and less error prone. Both EJB and Spring can use JTA under the covers.

---

170. Most database systems use shared (read) and exclusive (write) locks, attributed to specific locking elements (rows, tables). While physical locking is demanded by the SQL standard, the pessimistic approach might hinder scalability. Modern databases have implemented lightweight locking techniques, such as multiversion concurrency control.

---

171. The implicit database locking is hidden behind the transaction isolation level configuration. Each isolation level comes with a predefined locking scheme, aimed for preventing a certain set of data integrity anomalies.

READ COMMITTED uses query-level shared locks and exclusive locks for the current transaction modified data. REPEATABLE READ and SERIALIZABLE use transaction-level shared locks when reading and exclusive locks when writing.

---

172. Paired with a conversation-level repeatable read storage, optimistic locking can ensure data integrity without trading scalability.

JPA supports both optimistic locking and persistence context repeatable reads, making it ideal for implementing logical transactions.

---

173. JPA supports explicit locking for the following operations:

- finding an entity
- locking an existing persistence context entity
- refreshing an entity
- querying through JPQL, Criteria or native queries

---

174. The LockModeType contains the following optimistic and pessimistic locking modes:

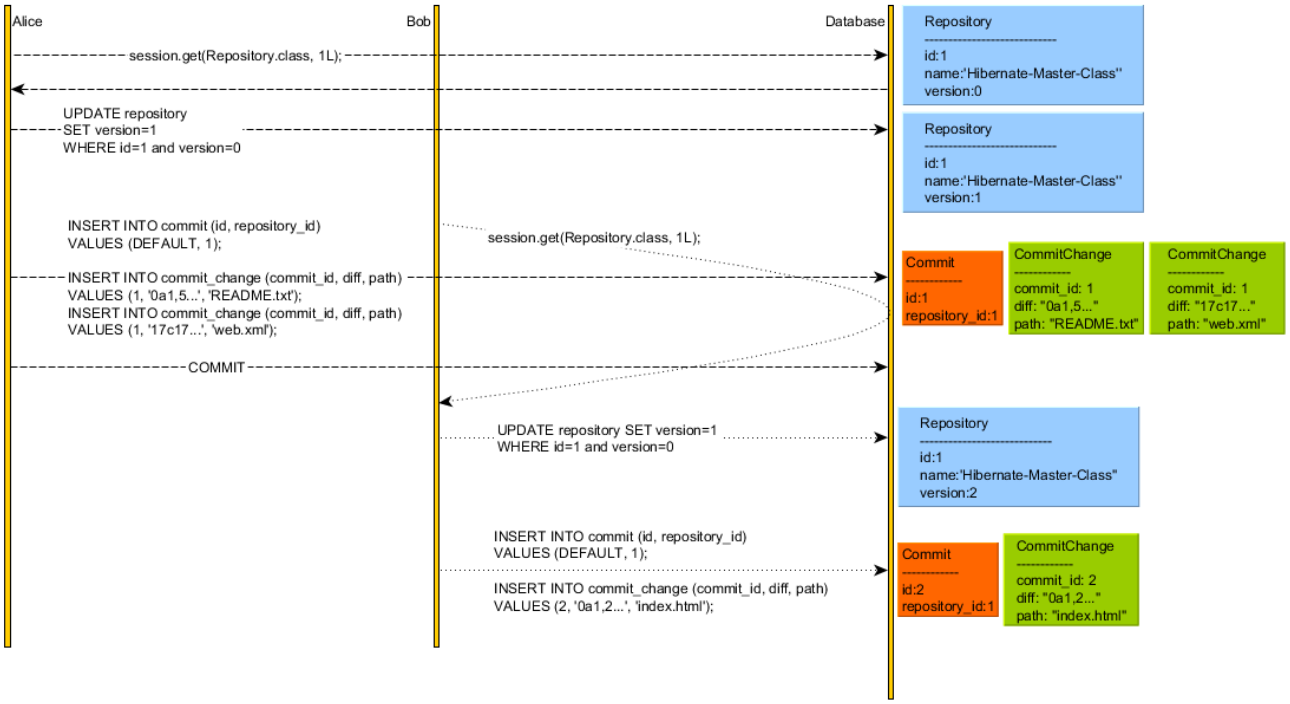
Lock Mode Type	Description
OPTIMISTIC_FORCE_INCREMENT	Always increases the entity version (even when the entity doesn't change) and issues a version check upon transaction commit, therefore ensuring optimistic locking repeatable reads.

https://vladmihalcea.files.wordpress.com/2015/02/explicitlockingoptimisticforceincrement1.png

NONE	In the absence of explicit locking, the application will use implicit locking (optimistic or pessimistic)
OPTIMISTIC	Always issues a version check upon transaction commit, therefore ensuring optimistic locking repeatable reads.

READ	Same as OPTIMISTIC.
WRITE	Same as OPTIMISTIC_FORCE_INCREMENT.
PESSIMISTIC_READ	A shared lock is acquired to prevent any other transaction from acquiring a PESSIMISTIC_WRITE lock.
PESSIMISTIC_WRITE	An exclusive lock is acquired to prevent any other transaction from acquiring a PESSIMISTIC_READ or a PESSIMISTIC_WRITE lock.
PESSIMISTIC_FORCE_INCREMENT	A database lock is acquired to prevent any other transaction from acquiring a PESSIMISTIC_READ or a PESSIMISTIC_WRITE lock and the entity version is incremented upon transaction commit.

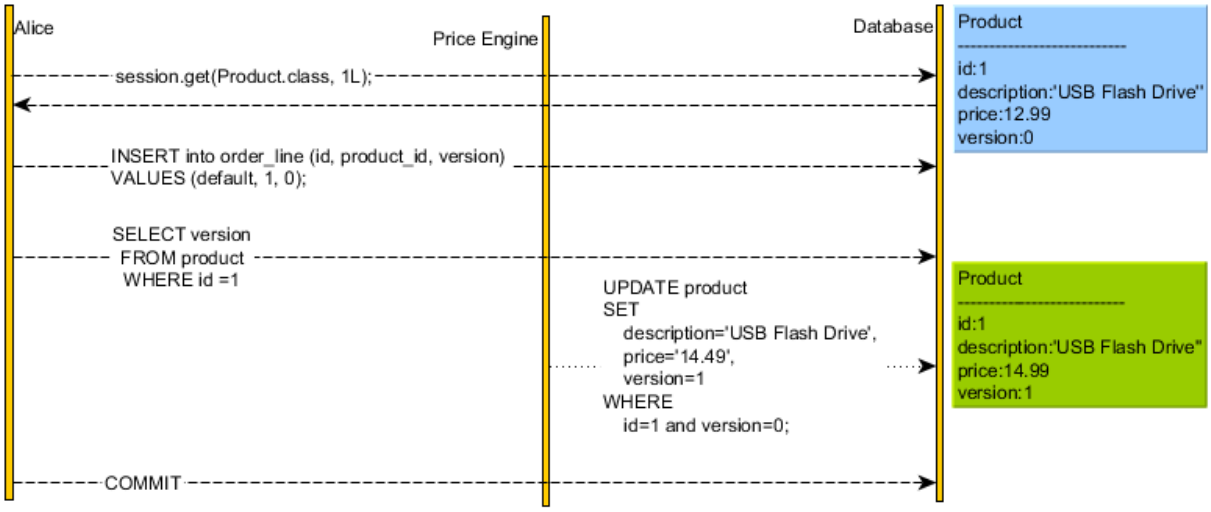
https://vladmihalcea.files.wordpress.com/2015/02/explicitlockingpessimisticforceincrement.png



---

175. Fix optimistic locking race conditions with pessimistic locking.

https://vladmihalcea.files.wordpress.com/2015/01/explicitlockinglockmodeoptimisticracecondition.png

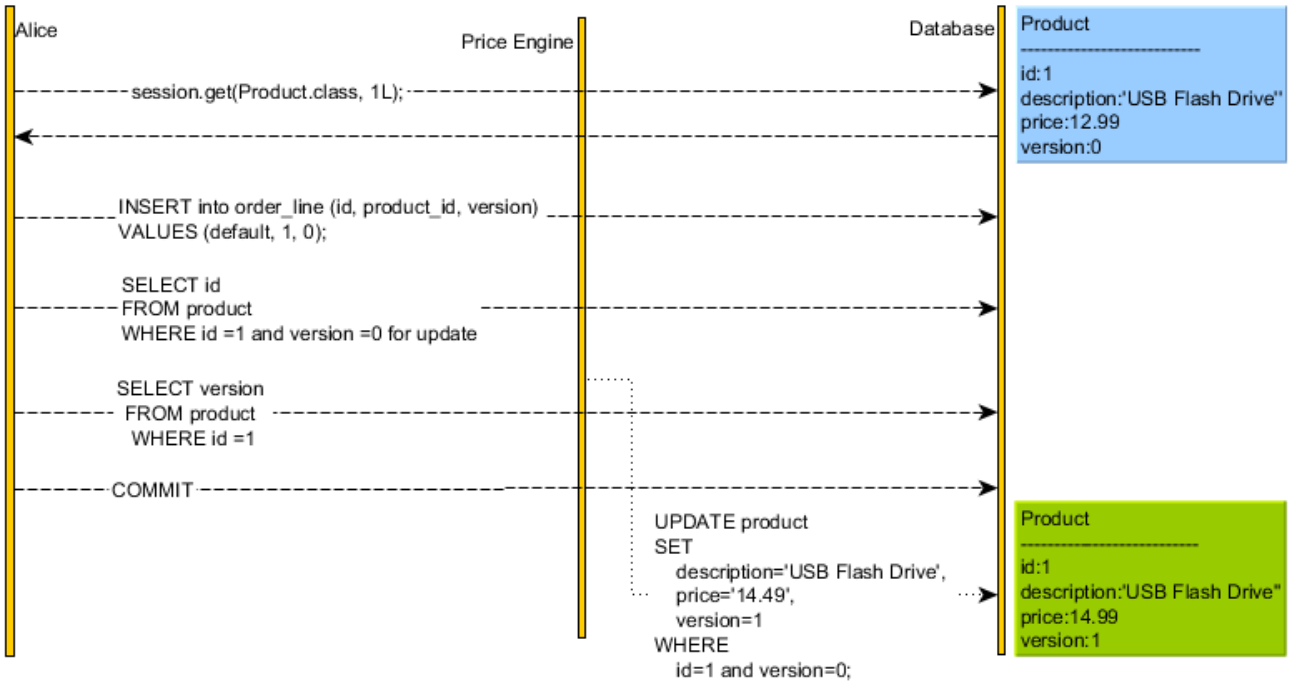


Alice fetches a Product
She then decides to order it
The Product optimistic lock is acquired
The Order is inserted in the current transaction database session

The Product version is checked by the Hibernate explicit optimistic locking routine
The price engine manages to commit the Product price change
Alice transaction is committed without realizing the Product price has just changed

FIXED via pesimistic locking (LockMode.PESSIMISTIC_READ):

https://vladmihalcea.com/2015/02/03/how-to-fix-optimistic-locking-race-conditions-with-pessimistic-locking/



---

176. Because a mutual exclusion locking would hinder scalability (treating reads and writes equally), most database systems use a readers-writer locking synchronization scheme, so that:

- A shared (read) lock blocks writers, allowing multiple readers to proceed
- An exclusive (write) lock blocks both readers and writers, making all write operations be applied sequentially

Java Persistence abstraction layer hides the database specific locking semantics, offering a common API that only requires two Lock Modes. The shared/read lock is acquired using the PESSIMISTIC_READ Lock Mode Type, and the exclusive/write lock is requested using PESSIMISTIC_WRITE instead.

---

177. Relational database systems use locks for preserving ACID guarantees, so it's important to understand how shared and exclusive row-level locks inter-operate. An explicit pessimistic lock is a very powerful database concurrency control mechanism and you might even use it for fixing an optimistic locking race condition.

---

178. Caching allows us to reuse a database response for multiple user requests.

---

179. The cache can therefore:

- reduce CPU/Memory/IO resource consumption on the database side
- reduce network traffic between application nodes and the database tier
- provide constant result fetch time, insensitive to traffic bursts
- provide a read-only view when the application is in maintenance mode (e.g. when upgrading the database schema)

---

180. The caching abstraction layer is aware of the database server, but the database knows nothing of the application-level cache. If some external process updates the database without touching the cache, the two data sources will get out of sync. Because few database servers support application-level notifications, the cache may break the strong consistency guarantees.

To avoid eventual consistency, both the database and the cache need to be enrolled in a distributed XA transaction, so the affected cache entries are either updated or invalidated synchronously.

---

181. The Persistence Context acts as a logical transaction storage, and each Entity instance can have

at-most one managed reference. No matter how many times we try to load the same Entity, the Hibernate Session will always return the same object reference. This behavior is generally depicted as the first-level cache.

---

182. A proper caching solution would have to span across multiple Hibernate Sessions and that's the reason Hibernate supports an additional second-level cache as well. The second-level cache is bound to the SessionFactory life-cycle, so it's destroyed only when the SessionFactory is closed (topically when the application is shutting down). The second-level cache is primarily entity-based oriented, although it supports an optional query-caching solution as well.

---

183. By default, the second-level cache is disabled and to activate it, we have to set the following Hibernate properties:

```
properties.put("hibernate.cache.use_second_level_cache", Boolean.TRUE.toString());
properties.put("hibernate.cache.region.factory_class",
               "org.hibernate.cache.ehcache.EhCacheRegionFactory");
```

---

184. The RegionFactory defines the second-level cache implementation provider, and the hibernate.cache.region.factory_class configuration is mandatory, once the hibernate.cache.use_second_level_cache property is set to true.

---

185. To enable entity-level caching, we need to annotate our cacheable entities as follows:

```
@Entity
@org.hibernate.annotations.Cache(usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
```

JPA also defines the @Cacheable annotation, but it doesn't support setting the concurrency strategy on entity-level.

---

186. The Session is always inspected first because it might already contain a managed entity instance. The second-level cache is verified before hitting the database, so its main purpose is to reduce the number of database accesses.

---

187. Every entity is stored as a CacheEntry, and the entity hydrated state is used for creating the cache entry value.

---

188. Hydration

In Hibernate nomenclature, hydration is when a JDBC ResultSet is transformed to an array of raw values:

```
final Object[] values = persister.hydrate(rs, id, object,
                        rootPersister, cols, eagerPropertyFetch, session);
```

---

189. The hydrated state is saved in the currently running Persistence Context as an EntityEntry object, which encapsulated the loading-time entity snapshot. The hydrated state is then used by:

- the default dirty checking mechanism, which compares the current entity data against the loading-time snapshot
- the second-level cache, whose cache entries are built from the the loading-time entity snapshot

---

190. The inverse operation is called dehydration and it copies the entity state into an INSERT or UPDATE statement.

---

191. Although Hibernate allows us to manipulate entity graphs, the second-level cache uses a disassembled hydrated state instead:

```
final CacheEntry entry = persister.buildCacheEntry(entity, hydratedState, version, session );
```

---

192. The hydrated state is disassembled prior to being stored in the CacheEntry:

```
this.disassembledState = TypeHelper.disassemble(
  state, persister.getPropertyTypes(),
persister.isLazyPropertiesCacheable()
       ? null : persister.getPropertyLaziness(), session, owner);
```

---

193. An example of a CacheEntry (The second-level cache elements):

```
Post post = new Post();
post.setName("Hibernate Master Class");

post.addDetails(new PostDetails());
post.addComment(new Comment("Good post!"));
post.addComment(new Comment("Nice post!"));
```

```
session.persist(post);
```

The Post entity has a one-to-many association to the Comment entity and an inverse one-to-one association to a PostDetails:

```
@OneToMany(cascade = CascadeType.ALL, mappedBy = "post")
private List<Comment> comments = new ArrayList<>();

@OneToOne(cascade = CascadeType.ALL, mappedBy = "post", optional = true)
private PostDetails details;
```

```
key = {org.hibernate.cache.spi.CacheKey@3855}
  key = {java.lang.Long@3860} "1"
  type = {org.hibernate.type.LongType@3861}
  entityOrRoleName = {java.lang.String@3862} "com.vladmihalcea.hibernate.masterclass.laboratory.cache.SecondLevelCacheTest$Post"
  tenantId = null
  hashCode = 31
          value = {org.hibernate.cache.spi.entry.StandardCacheEntryImpl@3856}
  disassembledState = {java.io.Serializable[3]@3864}
    0 = {java.lang.Long@3860} "1"
    1 = {java.lang.String@3865} "Hibernate Master Class"
  subclass = {java.lang.String@3862} "com.vladmihalcea.hibernate.masterclass.laboratory.cache.SecondLevelCacheTest$Post"
  lazyPropertiesAreUnfetched = false
  version = null
```

The CacheKey contains the entity identifier and the CacheEntry contains the entity disassembled hydrated state.

The Post entry cache value consists of the name column and the id, which is set by the one-to-many Comment association.

Neither the one-to-many nor the inverse one-to-one associations are embedded in the Post CacheEntry.

The PostDetails entity Primary Key is referencing the associated Post entity Primary Key, and it therefore has a one-to-one association with the Post entity.

```
@OneToOne
@JoinColumn(name = "id")
@MapsId
private Post post;
```

The second-level cache generate the following cache element:

```
key = {org.hibernate.cache.spi.CacheKey@3927}
  key = {java.lang.Long@3897} "1"
  type = {org.hibernate.type.LongType@3898}
  entityOrRoleName = {java.lang.String@3932} "com.vladmihalcea.hibernate.masterclass.laboratory.cache.SecondLevelCacheTest$PostDetails"
  tenantId = null
  hashCode = 31
          value = {org.hibernate.cache.spi.entry.StandardCacheEntryImpl@3928}
  disassembledState = {java.io.Serializable[2]@3933}
    0 = {java.sql.Timestamp@3935} "2015-04-06 15:36:13.626"
  subclass = {java.lang.String@3932} "com.vladmihalcea.hibernate.masterclass.laboratory.cache.SecondLevelCacheTest$PostDetails"
  lazyPropertiesAreUnfetched = false
  version = null
```

The disassembled state contains only the createdOn entity property, since the entity identifier is embedded in the CacheKey.

The Comment entity has a many-to-one association to a Post:

```
@ManyToOne
private Post post;
```

Hibernate generates the following second-level cache element:

```
key = {org.hibernate.cache.spi.CacheKey@3857}
  key = {java.lang.Long@3864} "2"
  type = {org.hibernate.type.LongType@3865}
  entityOrRoleName = {java.lang.String@3863} "com.vladmihalcea.hibernate.masterclass.laboratory.cache.SecondLevelCacheTest$Comment"
  tenantId = null
  hashCode = 62
```

```
            value = {org.hibernate.cache.spi.entry.StandardCacheEntryImpl@3858}
  disassembledState = {java.io.Serializable[2]@3862}
    0 = {java.lang.Long@3867} "1"
    1 = {java.lang.String@3868} "Good post!"
  subclass = {java.lang.String@3863} "com.vladmihalcea.hibernate.masterclass.laboratory.cache.SecondLevelCacheTest$Comment"
  lazyPropertiesAreUnfetched = false
  version = null
```

The disassembled state contains the Post.id Foreign Key reference and the review column, therefore mirroring the associated database table definition.

---

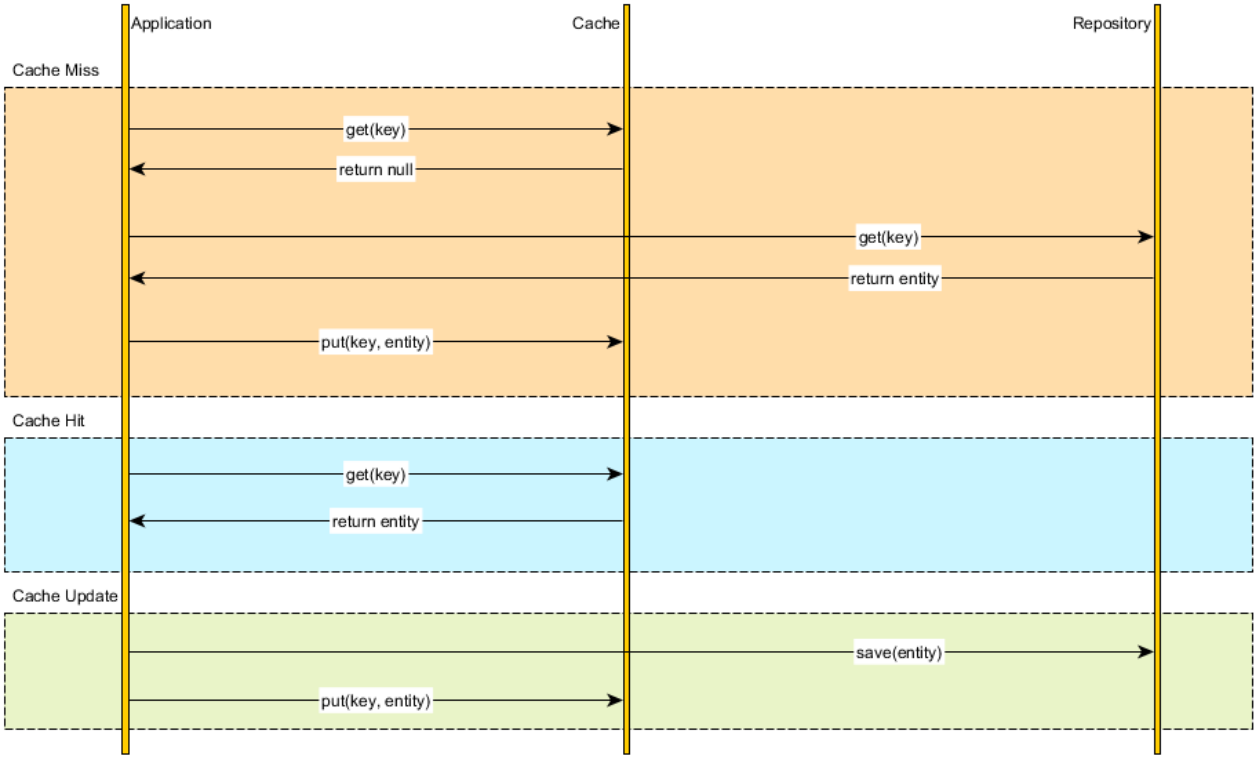
194. The second-level cache is a relational data cache, so it stores data in a normalized form, and each entity update affects only one cache entry. Reading a whole entity graph is not possible since the entity associations are not materialized in the second-level cache entries.

An aggregated entity graph yields better performance for read operations at the cost of complicating write operations. If the cached data is not normalized and scattered across various aggregated models, an entity update would have to modify multiple cache entries, therefore affecting the write operations performance.

Because it mirrors the underlying relation data, the second-level cache offers various concurrency strategy mechanisms so we can balance read performance and strong consistency guarantees.

---

195. Cache-aside (https://vladmihalcea.files.wordpress.com/2015/04/cacheaside.png) [cache-database synchronization strategy]

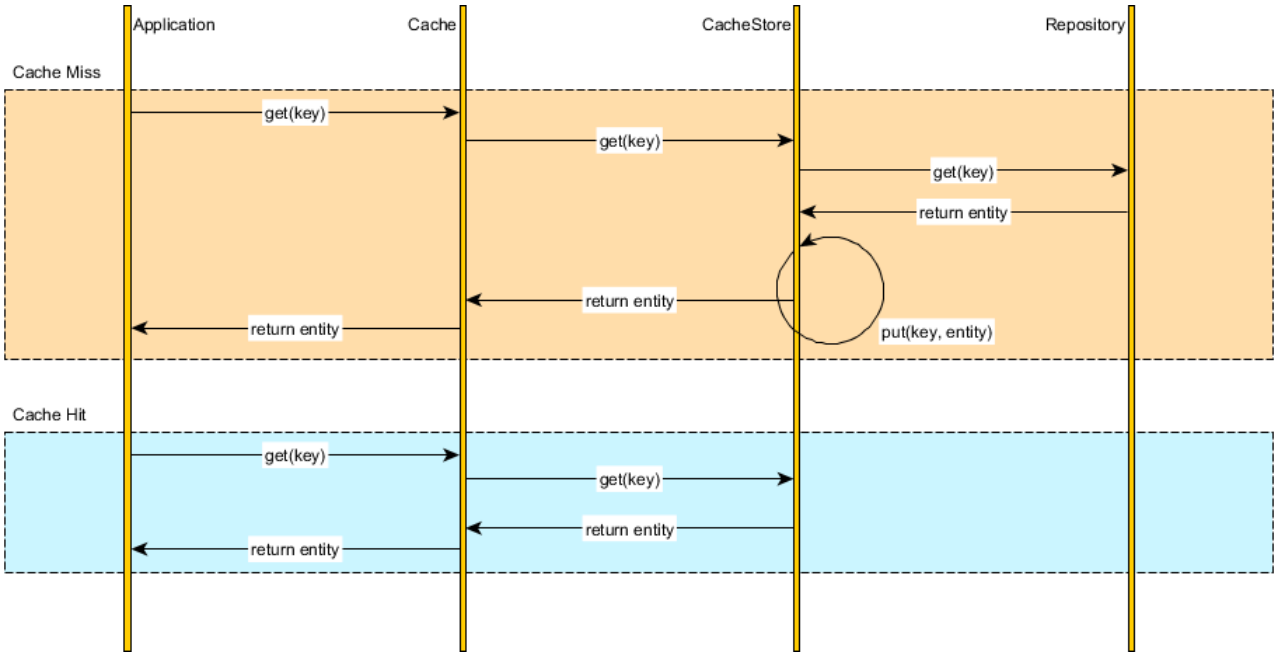


The application code can manually manage both the database and the cache information. The application logic inspects the cache before hitting the database and it updates the cache after any database modification.

Mixing caching management and application is not very appealing, especially if we have to repeat these steps in every data retrieval method. Leveraging an Aspect-Oriented caching interceptor can mitigate the cache leaking into the application code, but it doesn't exonerate us from making sure that both the database and the cache are properly synchronized.

---

196. Cache-aside (https://vladmihalcea.files.wordpress.com/2015/04/cachereadthrough.png) [cache-database synchronization strategy]

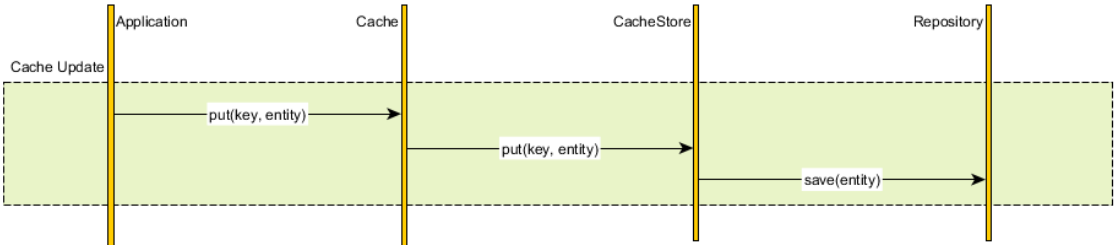


Instead of managing both the database and the cache, we can simply delegate the database synchronization to the cache provider. All data interactions is therefore done through the cache abstraction layer.

Upon fetching a cache entry, the Cache verifies the cached element availability and loads the underlying resource on our behalf. The application uses the cache as the system of record and the cache is able to auto-populate on demand.

---

197. Write-through (https://vladmihalcea.files.wordpress.com/2015/04/cachewritethrough.png) [cache-database synchronization strategy]



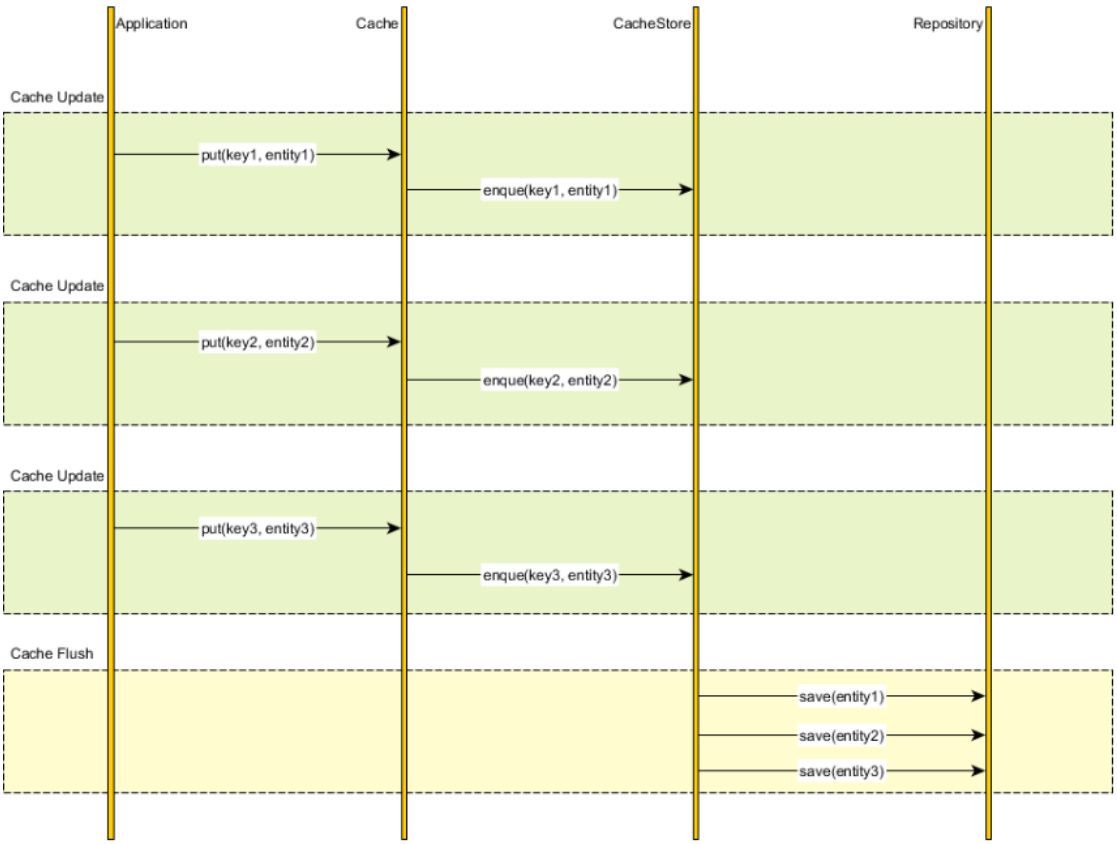
Analogous to the read-through data fetching strategy, the cache can update the underlying database every time a cache entry is changed.

Although the database and the cache are updated synchronously, we have the liberty of choosing the transaction boundaries according to our current business requirements.

If strong consistency is mandatory and the cache provider offers an XAResource we can then enlist the cache and the database in the same global transaction. The database and the cache are therefore updated in a single atomic unit-of-work.
If consistency can be weaken, we can update the cache and the database sequentially, without using a global transaction. Usually the cache is changed first and if the database update fails, the cache can use a compensating action to roll-back the current transaction changes.

---

198. Write-behind caching (https://vladmihalcea.files.wordpress.com/2015/04/cachewritebehind.png) [cache-database synchronization strategy]



If strong consistency is not mandated, we can simply enqueue the cache changes and periodically flush them to the database.

This strategy is employed by the Java Persistence EntityManager (first-level cache), all entity state transitions being flushed towards the end of the current running transaction (or when a query is issued).

Although it breaks transaction guarantees, the write-behind caching strategy can outperform the write-through policy, because database updates can be batched and the number of DML transactions is also reduced.

---

199. Hibernate READ_ONLY CacheConcurrencyStrategy (second-level caching)

If the cached data is immutable (neither the database nor the cache are able modify it), we can safely cache it without worrying of any consistency issues. Read-only data is always a good candidate for application-level caching, improving read performance without having to relax consistency guarantees. Read-only cache entries are not allowed to be updated. Any such attempt ends up in an exception being thrown. Because read-only cache entities are practically immutable it's good practice to attribute them the Hibernate specific @Immutable annotation. Read-only cache entries are removed when the associated entity is deleted as well. The remove entity state transition is enqueued by PersistenceContext, and at flush time, both the database and the second-level cache will delete the associated entity record.

All entities are cached as read-only elements:

```
@org.hibernate.annotations.Cache(
    usage = CacheConcurrencyStrategy.READ_ONLY
)
```

In Hibernate 4, the read-only second-level cache uses a read-through caching strategy, entities being cached upon fetching.

In Hibernate 5, READ_ONLY entities are write-through when using a SEQUENCE or a TABLE generator, while they are read-through for IDENTITY generator.

---

200. Collection caching

The Commit entity has a collection of Change components:

```
@ElementCollection
@CollectionTable(
    name="commit_change",
    joinColumns=@JoinColumn(name="commit_id")
)
private List<Change> changes = new ArrayList<>();
```

Although the Commit entity is cached as a read-only element, the Change collection is ignored by the second-level cache. Although the Commit entity is retrieved from the cache, the Change collection is always fetched from the database. Since the Changes are immutable too, we would like to cache them as well, to save unnecessary database round-trips.

Enabling Collection cache support

Collections are not cached by default, and to enable this behavior, we have to annotate them with the a cache concurrency strategy:

```
@ElementCollection
@CollectionTable(
    name="commit_change",
    joinColumns=@JoinColumn(name="commit_id")
)
@org.hibernate.annotations.Cache(
    usage = CacheConcurrencyStrategy.READ_ONLY
)
private List<Change> changes = new ArrayList<>();
```

Once the collection is cached, we can fetch the Commit entity along with all its Changes without hitting the database.

---

201. Read-only entities are safe for caching and we can load an entire immutable entity graph using the second-level cache only. Because the cache is read-through, entities are cached upon being fetched from the database. The read-only cache is not write-through because persisting an entity only materializes into a new database row, without propagating to the cache as well.

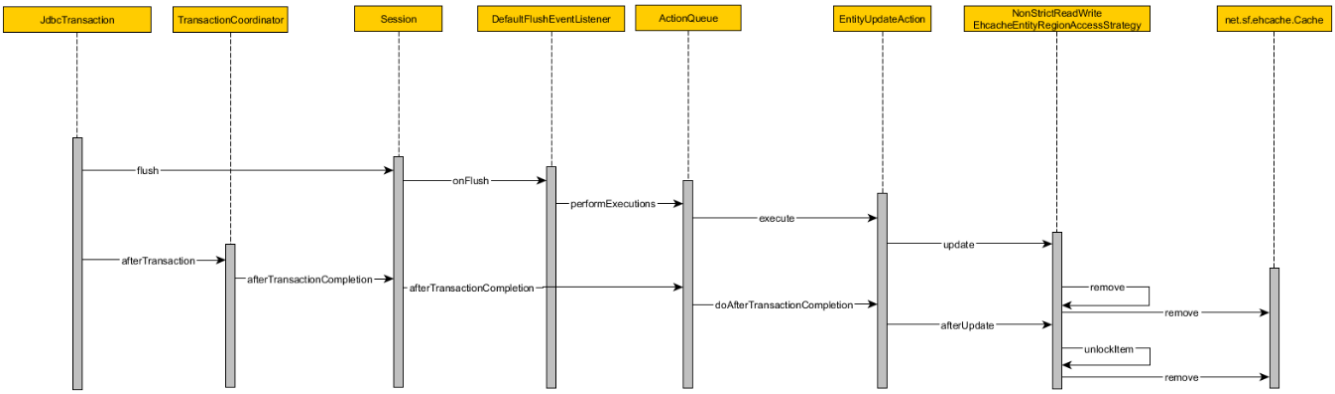
---

202. Hibernate NONSTRICT_READ_WRITE CacheConcurrencyStrategy (second-level caching)

When cached data is changeable, we need to use a read-write caching strategy as NONSTRICT_READ_WRITE second-level cache.

The NONSTRICT_READ_WRITE mode is not a write-though caching strategy but a read-through cache concurrency mode because cache entries are invalidated, instead of being updated. The cache invalidation is not synchronized with the current database transaction. Even if the associated Cache region entry gets invalidated twice (before and after transaction completion), there's still a tiny time window when the cache and the database might drift apart.

https://vladmihalcea.files.wordpress.com/2015/05/nonstrictreadwritecacheconcurrencystrategy4.png



First, the cache is invalidated before the database transaction gets committed, during flush time:

The current Hibernate Transaction (e.g. JdbcTransaction, JtaTransaction) is flushed
The DefaultFlushEventListener executes the current ActionQueue
The EntityUpdateAction calls the update method of the EntityRegionAccessStrategy
The NonStrictReadWriteEhcacheCollectionRegionAccessStrategy removes the cache entry from the underlying EhcacheEntityRegion
After the database transaction is committed, the cache entry is removed once more:

The current Hibernate Transaction after completion callback is called
The current Session propagates this event to its internal ActionQueue
The EntityUpdateAction calls the afterUpdate method on the EntityRegionAccessStrategy
The NonStrictReadWriteEhcacheCollectionRegionAccessStrategy calls the remove method on the underlying EhcacheEntityRegion

---

203. Optimistic concurrency control is an effective way of dealing with lost updates in long conversations and this technique can mitigate the NONSTRICT_READ_WRITE inconsistency issue as well.

The NONSTRICT_READ_WRITE concurrency strategy is a good choice for read-mostly applications (if backed-up by the optimistic locking mechanism). For write-intensive scenarios, the cache invalidation mechanism would increase the cache miss rate, therefore rendering this technique inefficient.

---

204. Hibernate READ_WRITE CacheConcurrencyStrategy (second-level caching)

NONSTRICT_READ_WRITE is a read-through caching strategy and updates end-up invalidating cache entries.
As simple as this strategy may be, the performance drops with the increase of write operations.
A write-through cache strategy is better choice for write-intensive applications, since cache entries can be undated rather than being discarded.

The READ_WRITE strategy is an asynchronous cache concurrency mechanism and to prevent data integrity issues (e.g. stale cache entries), it uses a locking mechanism that provides unit-of-work isolation guarantees.

---

205. Inserting data

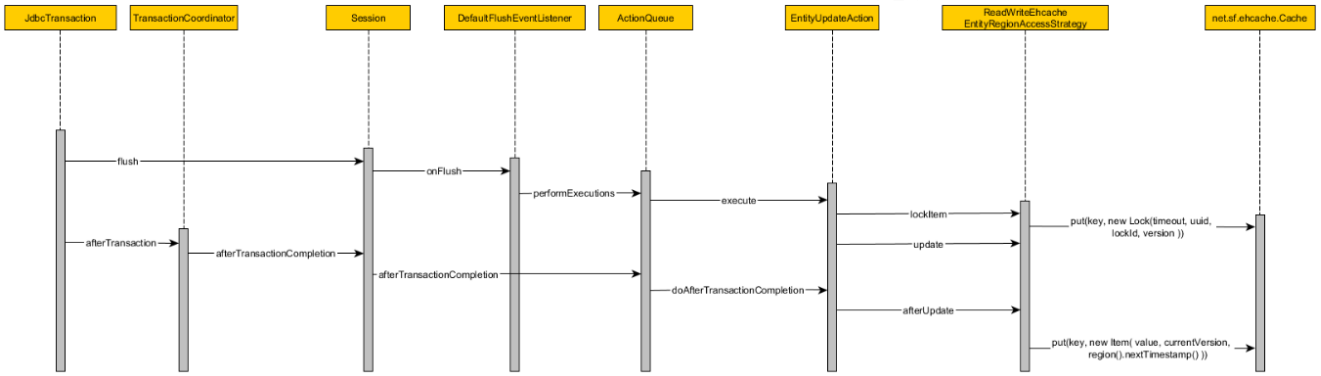
Because persisted entities are uniquely identified (each entity being assigned to a distinct database row), the newly created entities get cached right after the database transaction is committed

---

206. Updating data

While inserting entities is a rather simple operation, for updates, we need to synchronize both the database and the cache entry. The READ_WRITE concurrency strategy employs a locking mechanism to ensure data integrity.

https://vladmihalcea.files.wordpress.com/2015/05/readwritecacheconcurrencystrategy_update4.png



The Hibernate Transaction commit procedure triggers a Session flush
The EntityUpdateAction replaces the current cache entry with a Lock object
The update method is used for synchronous cache updates so it doesn't do anything when using an asynchronous cache concurrency strategy, like READ_WRITE
After the database transaction is committed, the after-transaction-completion callbacks are called
The EntityUpdateAction calls the afterUpdate method of the EntityRegionAccessStrategy
The ReadWriteEhcacheEntityRegionAccessStrategy replaces the Lock entry with an actual Item, encapsulating the entity dissembled state

---

207. Deleting data

Deleting entities is similar to the update process.

---

208. Timing out

If the database operation fails, the current cache entry holds a Lock object and it cannot rollback to its previous Item state. For this reason, the Lock must timeout to allow the cache entry to be replaced by an actual Item object. Unless we override the net.sf.ehcache.hibernate.cache_lock_timeout property, the default timeout is 60 seconds.

```
properties.put("net.sf.ehcache.hibernate.cache_lock_timeout", String.valueOf(250));
```

---

209. The READ_WRITE concurrency strategy offers the benefits of a write-through caching mechanism, but you need to understand it's inner workings to decide if it's good fit for your current project data access requirements.

For heavy write contention scenarios, the locking constructs will make other concurrent transactions hit the database, so you must decide if a synchronous cache concurrency strategy is better suited in this situation.

---

210. Hibernate TRANSACTIONAL CacheConcurrencyStrategy

While the READ_WRITE CacheConcurrencyStartegy is an asynchronous write-though caching mechanism (since changes are being propagated only after the current database transaction is completed), the TRANSACTIONAL CacheConcurrencyStartegy is synchronized with the current XA transaction.

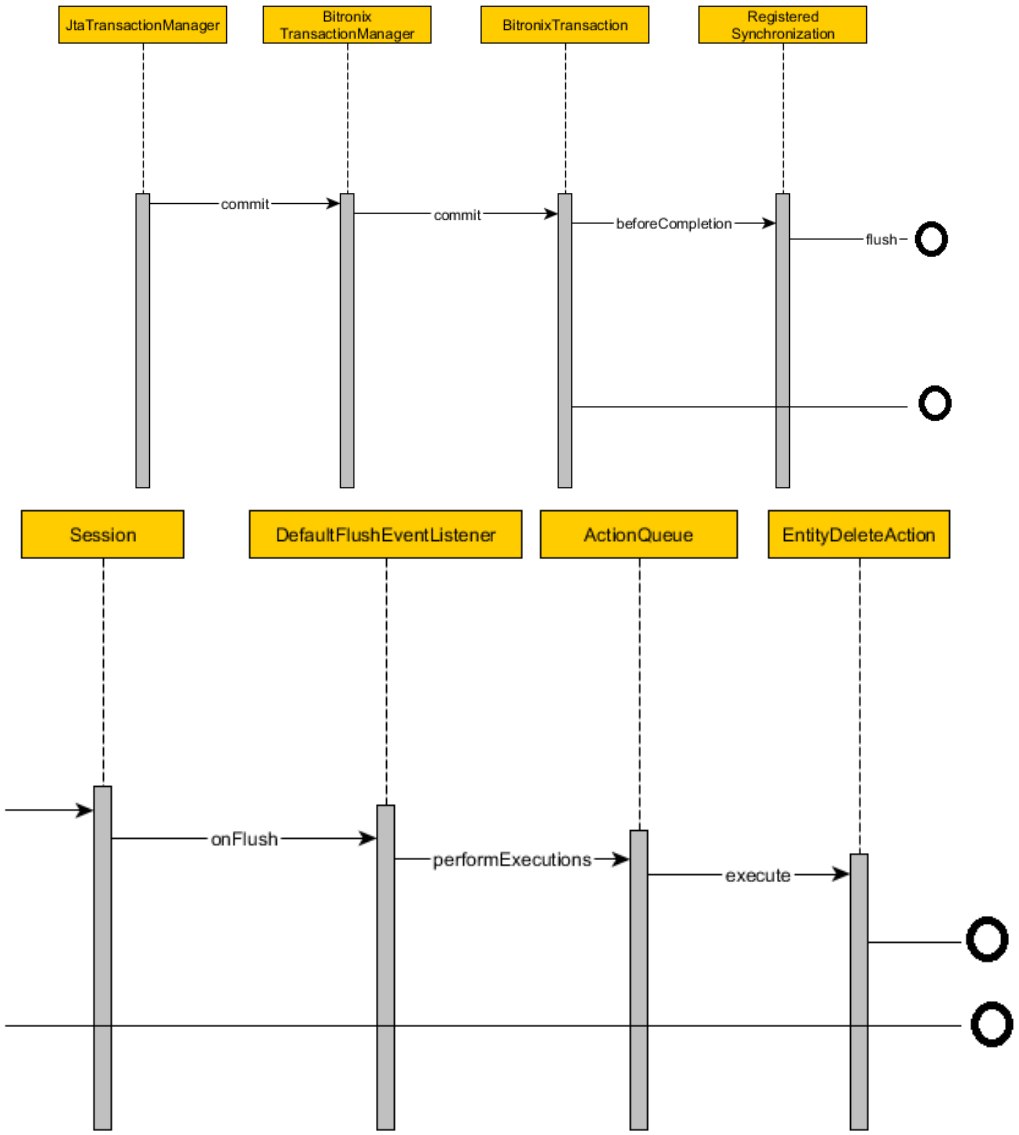
To enlist two sources of data (the database and the second-level cache) in the same global transaction, we need to use the Java Transaction API and a JTA transaction manager must coordinate the participating XA resources.

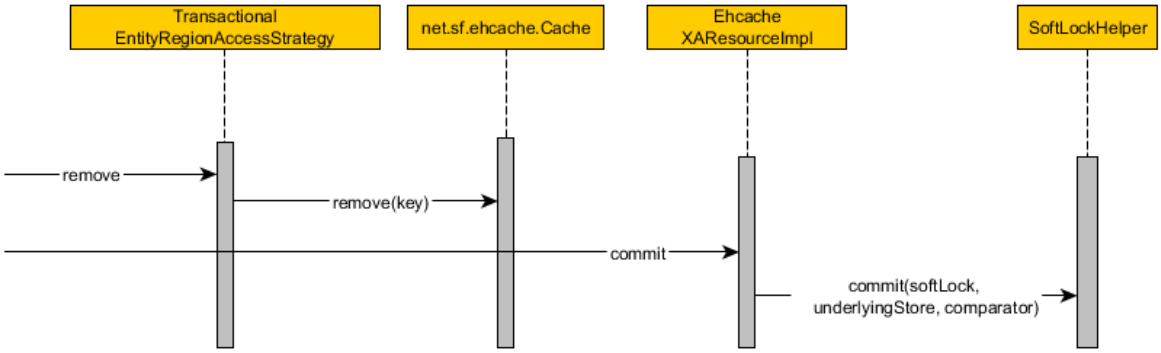
Is a good choice to use is Bitronix Transaction Manager, since it's automatically discovered by EhCache and it also supports the one-phase commit (1PC) optimization.

The EhCache second-level cache implementation offers two failure recovery options: xa_strict and xa.

---

211. xa_strict (https://vladmihalcea.files.wordpress.com/2015/05/transactionalxastrictcacheconcurrencystrategy.png)





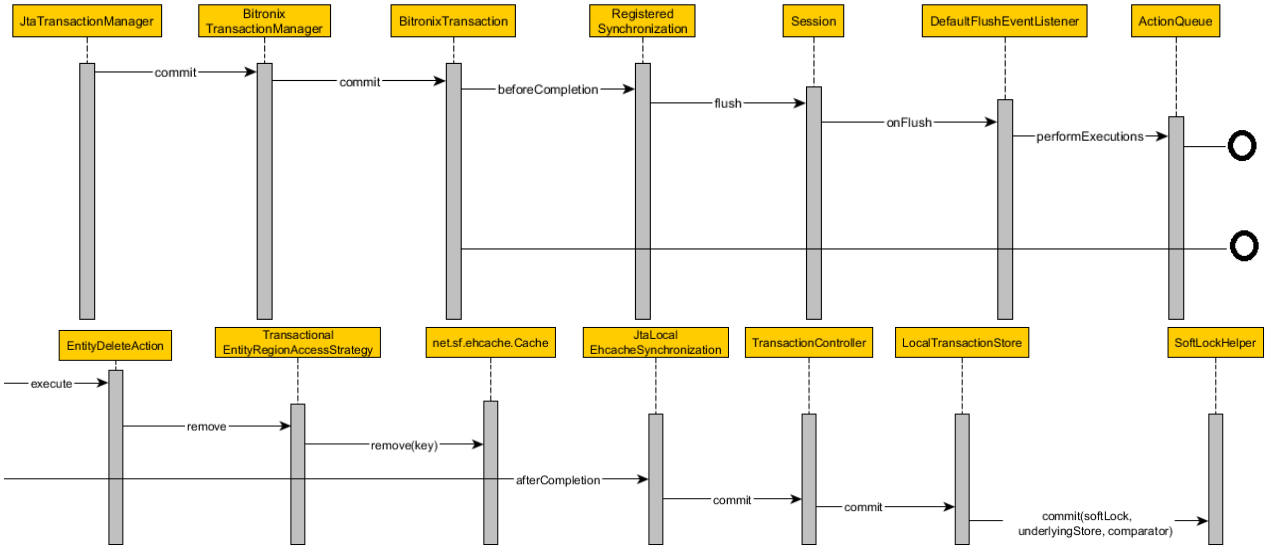
In this mode, the second-level cache exposes an XAResource interface, so it can participate in the two-phase commit (2PC) protocol.

The entity state is modified both in the database and in the cache, but these changes are isolated from other concurrent transactions and they become visible once the current XA transaction gets committed.

The database and the cache remain consistent even in case of an application crash.

---

212. xa (https://vladmihalcea.files.wordpress.com/2015/05/transactionalxacacheconcurrencystrategy.png)



If only one data source participates in a globaltransaction, the transaction manager can apply the one-phase commit optimization. The second-level cache is managed through a Synchronization transaction callback. The second-level cache doesn't actively participates in deciding the transaction outcome, as it merely executes according to the current database transaction outcome.

This mode trades durability for latency and in case of a server crash (happening in between the database transaction commit and the second-level cache transaction callback), the two data sources will drift apart. This issue can be mitigated if our entities employ an optimistic concurrency control mechanism, so even if we read stale data, we will not lose updates upon writing.

---

213. The TRANSACTIONAL CacheConcurrencyStrategy employes a READ_COMMITTED transaction isolation, preventing dirty reads while still allowing the lost updates phenomena. Adding optimistic locking can eliminate the lost update anomaly since the database transaction will rollback on version mismatches. Once the database transaction fails, the current XA transaction is rolled back, causing the cache to discard all uncommitted changes.

If the READ_WRITE concurrency strategy implies less overhead, the TRANSACTIONAL synchronization mechanism is appealing for higher write-read ratios (requiring less database hits compared to its READ_WRITE counterpart). The inherent performance penalty must be compared against the READ_WRITE extra database access, when deciding which mode is more suitable for a given data access pattern.

---

214. Hibernate Collection Cache

The Collection Cache is a very useful feature, complementing the second-level entity cache.
This way we can store an entire entity graph, reducing the database querying workload in read-mostly applications. Like with AUTO flushing, Hibernate cannot introspect the affected table spaces when executing native queries. To avoid consistency issues (when using AUTO flushing) or cache misses (second-level cache), whenever we need to run a native query we have to explicitly declare the targeted tables, so Hibernate can take the appropriate actions (e.g. flushing or invalidating cache regions).

---

215. Hibernate Query Cache

The Query Cache is strictly related to Entities, and it draws an association between a search criterion and the Entities fulfilling that specific query filter.

---

216. The Query Cache is disabled by default, and to activate it, we need to supply the following Hibernate property:

```
properties.put("hibernate.cache.use_query_cache", Boolean.TRUE.toString());
```

For Hibernate to cache a given query result, we need to explicitly set the cachable query attribute when creating the Query. Use setCacheable(true) in createQuery(...).setCacheable(true);

---

217. HQL/JPQL Query Invalidation

Hibernate second-level cache favors strong-consistency, and the Query Cache is no different. Like with flushing, the Query Cache can invalidate its entries whenever the associated table space changes. Every time we persist/remove/update an Entity, all Query Cache entries using that particular table will get invalidated.

---

218. Native Query Invalidation

As I previously stated, native queries leave Hibernate in the dark, as it cannot know which tables the native query might modify eventually.

---

219. Native Query Cache Region Synchronization

Hibernate allows us to define the query table space through query synchronization hints. When supplying this info, Hibernate can invalidate the requested cache regions:

```
session.createSQLQuery("update Author set name = '\"'||name||'\"' ").
                addSynchronizedEntityClass(Author.class).executeUpdate());
```

Only the provided table space was invalidated, leaving the Post (Foo) Query Cache untouched. Mixing native queries and Query Caching is possible, but it requires a little bit of diligence.

---

220. The Query Cache can boost the application performance for frequently executed entity queries, but it's not a free ride. It's susceptible to consistency issues and without a proper memory management control mechanism, it can easily grow quite large.

---

221. For a production environment, I always prefer using incremental DDL scripts, since I can always know what version is deployed on a given server, and which are the newer scripts required to be deployed. I've been relying on Flyway to manage the schema updates for me, and I'm very content with it.

---

222. Hibernate is very flexible when it comes to configuring it. Luckily we can customize the DDL generation using the "hibernate.hbm2ddl.auto" SessionFactory property.

The simplest way to deploy a schema is to use the "update" option. This is useful for testing purposes. I wouldn't rely on it for a production environment, for which incremental DDL scripts is a better approach.

So, choosing the "update" option is one choice for Integration Testing schema management.

---

223. persistence.xml sample (https://github.com/vladmihalcea/vladmihalcea.wordpress.com/tree/master/hibernate-facts)

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
             xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
                                 http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
    <persistence-unit name="testPersistenceUnit" transaction-type="JTA">
        <provider>org.hibernate.ejb.HibernatePersistence</provider>
        <exclude-unlisted-classes>false</exclude-unlisted-classes>
```

```
            <properties>
                <property name="hibernate.archive.autodetection" value="class, hbm"/>
                <property name="hibernate.transaction.jta.platform"
                          value="org.hibernate.service.jta.platform.internal.BitronixJtaPlatform" />
                <property name="hibernate.dialect" value="org.hibernate.dialect.HSQLDialect"/>
                <em><property name="hibernate.hbm2ddl.auto" value="update"/></em>
                <property name="hibernate.show_sql" value="true"/>
            </properties>
        </persistence-unit>
    </persistence>
```

And the dataSource configuration looks like:

```
    <bean id="dataSource" class="org.springframework.jdbc.
                                                datasource.TransactionAwareDataSourceProxy">
        <constructor-arg>
            <bean class="bitronix.tm.resource.jdbc.PoolingDataSource" init-method="init"
                  destroy-method="close">
                <property name="className" value="bitronix.tm.resource.jdbc.lrc.LrcXADataSource"/>
                <property name="uniqueName" value="testDataSource"/>
                <property name="minPoolSize" value="0"/>
                <property name="maxPoolSize" value="5"/>
                // this is needed by Hibernate, but we can set it to false after
                // we generate the DDL via hmb2ddl and load them eager
                <property name="allowLocalTransactions" value="true" />
                <property name="driverProperties">
                    <props>
                        <prop key="user">${jdbc.username}</prop>
                        <prop key="password">${jdbc.password}</prop>
                        <prop key="url">${jdbc.url}</prop>
                        <prop key="driverClassName">${jdbc.driverClassName}</prop>
                    </props>
                </property>
            </bean>
        </constructor-arg>
    </bean>
```

---

224. I think Bitronix is one of the most reliable tools I've ever worked with. When I was developing JEE applications, I was taking advantage of the Transaction Manager supplied by the Application Server in use. For Spring based projects, I had to employ a stand-alone Transaction Manager, and after evaluating JOTM, Atomikos and Bitronix, I settled for Bitronix. That was 5 years ago, and ever since I've deployed several applications, making use of it.

I prefer using XA Transactions even if the application is currently using only one Data Source. I don't have to worry about any noticeable performance penalty of employing JTA, as Bitronix uses 1PC (One-Phase Commit) when the current Transaction uses only one enlisted Data Source. It also makes possible of adding up to one non-XA Data Source, thanks to the Last Resource Commit optimization.

When using JTA, it's not advisable to mix XA and Local Transactions, since not all XA Data Sources allow operating inside a Local Transaction, so I tend to avoid this as much as possible.

Unfortunately, as simple as this DDL generation method is, it has one flaw which I am not too fond of. I can't disable the "allowLocalTransactions" setting since Hibernate creates the DDL script and updates it outside of an XA Transaction.

Another drawback is that you have little control over what DDL script Hibernate deploys on your behalf, and in this particular context I don't like compromising flexibility over convenience.

If you don't use JTA, and you don't need the flexibility of deciding what DDL schema would be deployed on your current database server, then the hibernate.hbm2ddl.auto="update" is probably your rightful choice.

---

225. Connection leaks should be detected during testing, therefore preventing connection leaks from occurring in a production environment.

---

226. Optional is not serializable therefore, an entity attribute should not be mapped as a java.util.Optional.

---

227. But just because we cannot map an entity attribute as Optional, it does not mean we cannot expose it using an Optional container. If we are using field-based access persistence, then the underlying entity attribute can be mapped using the actual persisted type, while the getter method can use an Optional instead.

```
@Entity(name = "PostComment")
@Table(name = "post_comment")
public static class PostComment
   implements Serializable {

   @Id
   @GeneratedValue
   private Long id;

   private String review;

   @ManyToOne(fetch = FetchType.LAZY)
   private Post post;

   @ManyToOne(fetch = FetchType.LAZY)
   private Attachment attachment;

   public Optional<Attachment> getAttachment() {
      return Optional.ofNullable(attachment);
   }

   public void setAttachment(Attachment attachment) {
      this.attachment = attachment;
   }

   //Other getters and setters omitted for brevity
}
```

If you're using property-based access, then the getter must expose the actual persisted type,
in which case you need to have a separate @Transient method that uses the Optional method return type.

We can process Attachment(s)as follows:

```
Attachment notAvailable = getNotAvaillableImage();

List<Attachment> attachments = comments.stream().map(pc -> pc.getAttachment()
            .orElse(notAvailable))
            .collect(Collectors.toList());
```

When using JPA and Hibernate, you can make use of the Java 1.8 Optional in your Domain Model entities.
However, you have to make sure not to use it as a persistent property type.
