Object-Relational Mapping

A persistence service is used for enterprise applications (e.g. written in Java) to handle persistent object data for long-term data management. In this exercise, we will first look at how general object-relational mapping is handled in this scenario and what key aspects of mapping are mainly focused.

1. Before we start to examine different aspects covered in object-relational mapping, let's first clarify the differences between two worlds, i.e. what is so-called object/relational impedance match?

A:

<table>
<thead>
<tr>
<th>objects</th>
<th>relations</th>
</tr>
</thead>
<tbody>
<tr>
<td>structure</td>
<td>relations</td>
</tr>
<tr>
<td>complex values, collections</td>
<td>flat tables</td>
</tr>
<tr>
<td>class hierarchies (inheritance)</td>
<td></td>
</tr>
<tr>
<td>relationships</td>
<td></td>
</tr>
<tr>
<td>binary</td>
<td>binary</td>
</tr>
<tr>
<td>1:1, 1:n, n:m (using collections)</td>
<td>1:1, 1:n</td>
</tr>
<tr>
<td>uni-/bi-directional references</td>
<td>value-based, symmetric</td>
</tr>
<tr>
<td>behavior</td>
<td></td>
</tr>
<tr>
<td>methods</td>
<td></td>
</tr>
<tr>
<td>access paradigm</td>
<td></td>
</tr>
<tr>
<td>object navigation (follow references)</td>
<td>declarative, set-oriented (queries)</td>
</tr>
</tbody>
</table>

2. When providing persistence in Java, three design principles/aspects of persistence needed to be considered. Explain these aspects of persistence:

A:

- **Orthogonal persistence:**
  a) persistence independent of data type, class
  b) instances of the same class may be transient or persistent.

- **Transitive persistence:** (aka persistence by reachability)
  a) objects can be explicitly designated to become persistent (i.e., roots)
  b) object referenced by persistent objects automatically become persistent, too

- **Persistence independence:** (aka transparent persistence)

---

a) code operating on transient and persistent objects is (almost) the same
b) "client object" side: no impact when interacting with persistent objects
   i. application may have to explicitly "persist" an object, but continues to
      use the same interface for interacting with the persistent object
   ii. interactions with a data store are not visible to/initiated by the client
        object, but happen automatically (e.g., when object state is modified or
        at EOT)

a) "persistent object" side: no special coding for "implementing" persistence

3. How is a single, primitive Java object (object class, object reference, instance of
   class, data types) mapped to its relational format in general? How does an entity
   mapping to RDBMS in JPA look like and which system components are involved?
   Explain whether JPA supports orthogonal persistence.

A:
   • Object class -> a single table or a set of multiple tables to support inheritance/complex
     field values
   • Object reference -> foreign key constraint
   • Entity instance/object -> one row in a table with a unique identifier
   • Data type mapping
   • In JPA: entities are annotated by @Entity. An entity must have primary key which is
     represented by annotation @Id. By default, field variable name is used as mapped column
     name whereas the annotation @Column can be used to explicitly set the mapped column
     name. Relationships between parent objects and containing objects are explained in the fol-

4. In the Entity-Relation Model², there is a concept called weak entity. Explain how
   are weak entities supported in JPA and how are they handled when mapped to
   RDBMS.

A: Embeddable class is used to support weak entities in JPA. Instances of an embeddable class are
   seen as embedded objects and do not have a persistent identity. Single-valued/collection-valued em-
   bedded objects are mapped to the same table as the containing entity, or to a collection table, re-
   spectively.

5. Inheritance is a key feature used in object-oriented (OO) programming model.
   Taking a login system example shown below, explain those three approaches intro-
   duced in the lecture which support mapping OO-inheritance to RDBMS. Fur-
   thermore, please discuss their advantages/disadvantages.

A:
   • Single table with discriminator column (InheritanceType.SINGLE_TABLE): stores columns for
     all attributes of any class in the hierarchy with a special discriminator column identifying
     the class within the hierarchy.
     +: avoid redundancy; each entity is stored as one tuple.

Middleware for Heterogeneous and Distributed Information Systems

+: avoid union and join operation at runtime to retrieve instances from super- and sub-class tables
-: increased storage cost (discriminator column, null values for sub-class specific columns)

- Vertical partitioning (InheritanceType.JOINED): one table per entity class with columns for newly defined attributes specific to the sub-class plus ID column.
  +: reduced storage cost
  -: increased update and search cost
  -: high runtime latency due to instance construction using join-operation on id column.

- Horizontal partitioning (InheritanceType.TABLE_PER_CLASS): one table per entity class, with columns for all attributes including inherited attributes from super-class.
  +: small storage cost; each entity is stored as one tuple and no additional null values as compared to SINGLE_TABLE approach.
  -: sub-class instance access can lead to recursive search in sub-classes.
  -: super-class instance access can lead to union of rows based on overlapping columns from both super- and sub-tables.

![User class hierarchy of login system](image)

Figure 1: Sample class hierarchy of login system

6. Relationships between either objects (Java) or relations (RDBMS) are differently supported in both data models. In Java, so-called "networks" of objects are normally used, i.e. an object contains another object using references and objects can be navigationally accessed. In relational data model, relationships are represented as value-based, primary key/foreign key relationships.

6.1. How many relationship types do we have? How many directionalities? How many combinations between relationship types and directionalities do we have? How are these combinations represented at object-oriented level? Explain the roles of owning side and inverse side in each combination.

A:

- Relationship types: one-to-one, one-to-many, many-to-one, many-to-many
- Directionalities: uni-/bidirectional
- Seven combinations. This is because a bidirectional one-to-many relationship is equivalent to a bidirectional many-to-one relationship.
they are represented using containing relationship. For instance, many students attend a lecture. Thus, a lecture object may contain a list of student object references.

bi-directional combination has a designated owning side and inverse side; for 1:n and n:1, the "many" side has to be the owning side which determines the state at the persistent data store. Uni-directional relationship only has an owning side.

6.2. Take an example that one person (firstname, lastname) has only one address (city, zipcode). How is this relationship mapped to database schema in all directionality settings in JPA?

A: Mapping is accomplished by using primary key/foreign key relationships. Table for the designated "owning" side, which is Person class, has to contain the foreign key.

Person(person_id, firstname, lastname, address_id); Address(address_id, city, zipcode)

6.3. Take the following customer-order relationship as an example. How is this mapped to relational schema?

A: This is a bi-directional one-to-many relationship. As defined previously the owning side is the "many" side which is Order class. Furthermore, mapped foreign key should also be covered in "owning" side.

Customer(customer_id); Order(order_id, name, customer_id)

6.4. With another order-lineitem relationship as below, how is this relationship again mapped to relational schema?

A: This is an exceptional case where a uni-directional one-to-many relationship. The owning side is Order class while the inverse side is LineItem class. However, in this case, the foreign key is mapped not onto the "owning" side but onto the "n" side which is LineItem class.

Exception: for uni-directional 1:n-relationship, foreign key is on the table for the "n" side.

Order(order_id, name); LineItem(item_id, comment, order_id)

6.5. Given an example that one student can attend multiple lectures while one lectured have multiple students, how is this case mapped now?

A: We need an extra join table for this many-to-many relationship which contains a composite key consisting of the primary keys from both student and lecture tables.

Student(student_id, name); Lecture(lecture_id, title); Jointable(student_id, lecture_id)
Entity Management

In JPA, programmers use annotations to specify the persistence properties so that entity managers can control the persistent state of entities at runtime.

1. Illustrate the life cycle of entity and explain the life cycle state and corresponding transition methods.

A:

```
new

managed

new()

refresh()
persist()
remove()

managed

Persistence context
ends

managed

merge()
persist()

removed

detached

```

Figure. Entity life cycle

The life of an entity instance has two main aspects: its relationship to a specific persistence context, and the synchronization of its state with the database. The EntityManager distinguishes between four states in the life cycle of an entity:

**New.** The entity instance was created in memory, but is not yet associated with either a persistent identity in the database or a persistence context. This is the state that an entity is in right after its creation using its constructor method. Changes in the entity state are not synchronized with the database at this stage.

**Managed.** The entity has a persistent identity in the database and is currently associated with a persistence context. This entity is in the managed state after the `persist()` method is called. Changes to the entity will be synchronized with the database when transactions are committed (whenever a business method returns) or when synchronization is explicitly triggered using the `flush()` operation.

**Detached.** The entity does have a persistent identity but is not or is no longer associated with the persistence context and no synchronization with database will happen. An entity becomes detached when the persistence context ends and can be synchronized with database again by merging it into a new persistence context.

**Removed.** The entity is currently associated with a persistence context but has been scheduled for removal from the database.

2. JPA uses the `cascade` annotation to support transitive persistence. Please first explain possible "cascade" types in JPA and examine what kind of underlying database facility could be utilized to support "all" or "remove" from cascade types.

A:

- cascade types: persist, merge, remove, refresh, all

---

3. Explain possible loading schemes. How are they supported in JPA?

A:
- "lazy loading" [Annotation fetch=FetchType.LAZY] - object is retrieved only when accessed based on primary key or reference (relationship); easy to implement; may cause increased communication with data source, resulting in performance drawbacks.
- "eager loading" [Annotation fetch=FetchType.EAGER] - when an object is requested, transi-
tively load all the objects reachable through references; requires construction/generation of complex data store queries; may cause a lot of unnecessary objects to be loaded.

4. How many approaches do we have to lookup entities? Compare their usage cases and explain their JPA support.

A:
- by entity persistent identifier/primary key (look up) - a single entity is returned by calling find(entityClass, primaryKey).
- by query - but in many situations we either don't know the primary key, or we need more than one result, or need to specify one or more search conditions. The method createQuery(queryString) returns java.persistence.Query object to retrieve sets of re-
sults by calling query.getResultList() subsequently.

5. Consider the abstract persistence schema depicted in 4. Give EJB QL statements for the following queries.

a. Find all lectures.
   SELECT l
   FROM Lecture l

b. Given a lecture name, find all students that attend lectures of that name.
   SELECT DISTINCT s
   FROM Student s, IN (s.lecture) AS l
   WHERE l.title = ?1

c. Given a student id, find all professors that teach this student.
   SELECT DISTINCT p
   FROM Professor p, IN (p.lecture) AS l, IN (l.student) AS s
   WHERE s.id = ?1
**Transaction and Concurrency control**

Persistent objects are manipulated in the scope of a transaction while the persistence is supported by relational databases. Now we discuss transaction support in JPA in a scenario where multiple applications/transactions run with persistent data.

1. What kind of anomaly can occur when an object that was detached is re-attached in the same application? How can this problem be compensated?
   
   A: 
   
   *Lost updates anomaly can occur in this case due to concurrent updates on persistent state and it can be controlled by explicit refresh or using optimistic locking.*

2. JPA supports optimistic locking with "read committed" as the highest isolation level. What kind of anomalies can still occur?
   
   A: 
   
   *Non-repeatable or phantom reads*

3. Comment on "possible load schemes from the persistence service might impact transaction processing differently when "read committed" is considered as the transaction isolation level."

   A: A parent entity contains a set of dependent entities and eagerly loaded each time when the parent entity is requested. Suppose a transaction that modifies an attribute (not its dependent entities) occurs concurrently with another transaction that modifies one of the dependent entity. In the eager-loading scheme, once the "parent" transaction wants to commit, the version number of dependent entities differs, resulting in rolling-back of the transaction running on the parent object, which is not expected from user.

---