7

JPQL

Java Persistence Query Language, SQL, and bulk operations

This chapter is focused on the Java Persistence Query Language (JPQL). After you are briefly introduced to the JPQL, you will jump right into creating queries. All aspects of a query are covered in this chapter, including the **`fetch join`** *operator. The* **`fetch join`** *operator enables you to eagerly fetch lazy relationships, potentially eliminating the* **`LazyInitializationException`***, which has plagued many applications that use an ORM solution. You will also look at the JPQL's bulk operation support. By the end of this chapter you will be familiar with all aspects of the JPQL.*

Overview

The JPQL is a database-independent, entity-based query language. It is an extension of the EJB Query Language (EJB QL) and adds many features that have been unavailable in the EJB QL. The JPQL includes support for projection (selecting individual entity fields instead of the entire entity), bulk operations (update and delete), subqueries, join, group by, and having operations. All JPQL operations are supported in both static (named queries) and dynamic queries. In addition, JPQL queries are polymorphic, so fetching **`Rank`** entities will return **`SpecialRank`** and **`PostCountRank`** instances (**`Rank`** is an abstract class, and JPQL returns only concrete types).

The JPQL is SQL-like in its syntax: "**`select from [where] [group by] [having] [order by]`**". If you know SQL, you already know much of the JPQL syntax. The JPQL also supports SQL-like functions such as max and min, but it is not as comprehensive as its SQL counterpart. Regardless of how similar, an important difference exists between the two: the JPQL operates on the "abstract persistence schema" and not on the physical schema defined in the database. You define the abstract persistence schema using JPA metadata in annotations or in ORM

files (see *Chapter 2: Entities Part I*, *Chapter 3: Entities Part II*, and *Chapter 4: Packaging* for more information about annotations and ORM). The abstract persistence schema encompasses entities, their fields, and any relationships you have defined. The JPA persistence provider translates your JPQL into native SQL queries, which means that you specify an entity or its fields in the **`SELECT`** portion of your query and entities in the **`WHERE`** clause. You can also use the dot (.) notation to traverse relationships in the **`SELECT`** portion of your query. Here is an example of traversing the **`PrivateMessage`** to **`User`** association from the **`agoraBB`** object model:

```
SELECT p.toUser FROM PrivateMessage p
```

This query will retrieve the **`User`** the **`PrivateMessage`** was addressed to.

Although the JPQL refers to all possible operations as queries, only three types of "queries" exist in reality: **`SELECT`** queries (what you would normally think of as a query), update queries (which are not queries but update statements), and delete queries (again not really queries but delete statements). The JPA specification refers to all of these operations as queries, and you define all three of them using the same syntax; all that differs is their effect on the database.

This chapter covers the following topics:

- JPQL overview
- Joins
- **`Where`**, **`Group by`**, **`Having`**
- The **`SELECT`** clause
- **`Order By`**
- Bulk operation support
- Examples

JPQL OVERVIEW

JPQL operates over what is called the *abstract persistence schema*, which is the schema that is defined by the entities of a persistence unit. With SQL, you select columns from database tables; with JPQL, you select fields from entities. All portions of a JPQL statement operate on entities and their fields or properties. You never refer to the database table or column an entity is mapped to. Here is the most basic JPQL statement you can create:

```
SELECT u FROM User
```

This query fetches all **`User`** instances from the database.

JPQL TYPES

JPQL is a typed language in which each expression in a JPQL statement has a type.

For example, take the following query:

```
SELECT u.username FROM User u
```

In this query, **`u.username`** is an expression that results in a **`String`** type. This query returns a list of **`String`** objects, and each item in the list represents a different user name in the system. The JPQL specification refers to the various types contained in an entity (fields and properties) as the *abstract schema type*.

An entity can have one or more of the following abstract schema types:

- state-field: This type includes any field or property of an entity that does not represent a relationship. The type of the field or the result of the **`get`** method for a property determines the abstract schema-type of a state-field.
- association-field: This type includes any field or property of an entity that represents a relationship. The abstract schema type of the association-field is that of the target entity.

The domain of your JPQL statement is defined by the entities of the persistence unit you are executing the query on.

An entity is referred to by its name in a query. An entity's name is the name you specified with the **`@Entity`** annotation or the entity's unqualified name. Consider the entity in Listing 7.1:

NOTE: JPQL queries are case insensitive, except for entity names and entity fields. Therefore, you can use any case you want in all other portions of the query. Throughout this chapter, I use all caps for JPQL-reserved identifiers, but you do not need to use this format in your application.

NOTE: A *typed language* is one that defines which operations are valid for which types. For example, if you try to compile "123" + 3 in Java, you receive an incompatible types compiler error.

Listing 7.1

```
package com.sourcebeat.jpa.model;

@Entity(name="message")
public class PrivateMessage extends ModelBase {

    // fields/methods removed for readability

}
```

The name of the entity in Listing 7.1 is "**message**" because you defined the name using the **@Entity** annotation (entity names must be unique within a persistence unit). In Listing 7.2, the entity's unqualified name is **User** (starts with an uppercase letter "**U**"). The entity's full name is **com.sourcebeat.jpa.model.User**, but its unqualified name is simply **User**.

Listing 7.2

```
package com.sourcebeat.jpa.model;

@Entity
public class User extends ModelBase {

    // fields/methods removed for readability

}
```

When referring to an entity in JPQL, you need to use the entity's name as specified with **@Entity** or its unqualified name. Some JPA persistence providers (for example, Hibernate) allow you to use the fully qualified name but other persistence providers (for example, TopLink) throw an exception. For portability, use the unqualified entity name in queries.

An identification variable is an identifier that is specified in a **FROM** clause. In the following query, **u** is an identification variable. The type of **u** is the abstract schema type of the entity that was identified by the name **User**. In this example, the entity is **User** (defined earlier), so the type of **u** is **User**.

```
SELECT u FROM User u
```

Identification variables can also be defined using the **JOIN** keyword, like this:

```
SELECT r.name FROM User u JOIN u.roles r
```

Here, the identification variables are **u** and **r**. **r** represents any **Role** that is directly reachable from a **User** instance.

Reserved identifiers

The JPA defines the following reserved identifiers (even though the list below is shown in uppercase, reserved identifiers are case insensitive, so **SELECT** is the same as **select** or **SelECt**):

SELECT, FROM, WHERE, UPDATE, DELETE, JOIN, OUTER, INNER, LEFT, GROUP, BY, HAVING, FETCH, DISTINCT, OBJECT, NULL, TRUE, FALSE, NOT, AND, OR, BETWEEN, LIKE, IN, AS, UNKNOWN, EMPTY, MEMBER, OF, IS, AVG, MAX, MIN, SUM, COUNT, ORDER, BY, ASC, DESC, MOD, UPPER, LOWER, TRIM, POSITION, CHARACTER_LENGTH, CHAR_LENGTH, BIT_LENGTH, CURRENT_TIME, CURRENT_DATE, CURRENT_TIMESTAMP, NEW, EXISTS, ALL, ANY, SOME.

UNKNOWN is not currently used in the JPQL.

Path expressions

A path expression is defined as an identification variable followed by the navigation operator (**.**) followed by a state-field or an association-field. You can traverse relationships as long as they are not collections. Figure 7.1 shows a portion of the **agoraBB** object model and is used to help explain path expressions.

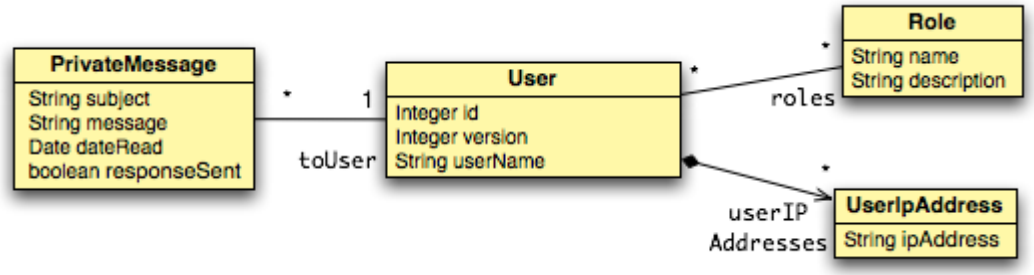

Figure 7.1: Path expression example model

The **PrivateMessage** to **User** relationship is represented by the **PrivateMessage** association-field **toUser**. **User** has a collection association-field to **Roles** called **roles** and another to **UserIpAddress** called **userIpAddress**. Given these relationships, you can perform the following navigations:

- **PrivateMessage** to **User** to **UserIpAddresses**

```
SELECT p.toUser.userIPAddresses from PrivateMessage p
```

- **PrivateMessage** to **User** to **Role**

```
SELECT p.toUser.roles from PrivateMessage p
```

- **User** to **Role**, fetching all distinct role names

```
SELECT DISTINCT r.name FROM User u JOIN u.roles r
```

- **User** to **UserIpAddress**

```
SELECT DISTINCT u.userIPAddresses FROM User u
```

The relationships in Figure 7.2 are slightly different than those shown in Figure 7.1.

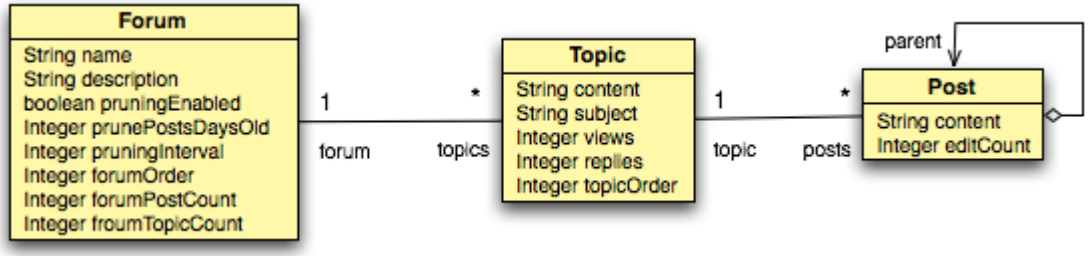

Figure 7.2: Path expression example model 2

In Figure 7.2, you have the **Forum** to **Topics** relationship represented by the **Forum** field **topics**, and the **Topics** to **Post** relationship represented by the **Topic** field **posts**. This object graph illustrates which JPQL path expressions are legal and which are not:

- **Forum** to **Topic**, fetching the subject: This path expression is illegal: you cannot navigate across collections.

```
SELECT f.topics.subject FROM Forum f - ILLEGAL
```

- **Forum** to **Topic** to **Post**: This path expression is legal using the **JOIN** operator.

```
SELECT t.subject FROM Forum f JOIN f.topics AS t
```

- **Forum** to **Topic** to **Post**: This path expression is illegal; you cannot navigate across a collection.

```
SELECT f.topics.posts FROM Forum f - ILLEGAL
```

- **Forum** to **Topic** to **Post**: This path expression is legal using the **JOIN** keyword.

```
SELECT p FROM Forum f JOIN f.topics t JOIN t.posts p
```

In summary, you can use the navigation operator (**.**) to traverse your entity object graph. The query type is determined by the variables in the **SELECT** clause. The **SELECT** clause can contain identification variables or path expressions. A path expression can navigate across the entity object graph as long as you move from left to right across single-value association-fields. You cannot navigate across a collection association-field or a state-field.

NOTE: Although the JPA specification states that you cannot navigate a collection association-field, my testing with both Hibernate and TopLink allowed me to navigate across a collection association and access a state-field in the target entity. The JPA BNF grammar for path navigation also indicates the **Forum** to **Topic**, so fetching the subject should be an illegal operation. If you need to access the target of a collection association-field, use the **JOIN** syntax shown in the second example.

RANGE VARIABLES

Range variables use syntax similar to SQL and tie together an entity name with an identification variable. A range variable declaration is defined (in the **FROM** clause of a query) as:

```
entityName [AS] identification_variable
```

You can use the same entity name in multiple-range variable declarations as in the example in Listing 7.3, which is taken from the JPA specification:

Listing 7.3

```
SELECT DISTINCT o1
FROM Order o1, Order o2
WHERE o1.quantity > o2.quantity AND
o2.customer.lastname = 'Smith' AND
o2.customer.firstname= 'John'
```

This query retrieves all orders that have a quantity greater than John Smith's order.

JOINS

A join occurs when the results of two or more entities are combined to produce the results of a JPQL query. JPQL joins are similar to SQL joins: In the end, all JPQL queries are translated into native SQL anyway. A join occurs when any of the following conditions are met:

- A path expression, which traverses an association-field, appears in the **SELECT** clause
- The join reserved identifier appears in the **WHERE** clause
- Two or more range variables are defined

If you have more than one entity involved in your query and you do not use a join, you retrieve all instances from all entities. This result is referred to as the *Cartesian product*. Assuming you have eight roles and four users in your system, the following query returns 32 objects:

```
SELECT r, u FROM Role r, User u
```

Typically, you will use some form of a join to reduce the number of entities returned from your query. If you want to join entities on a field other than their primary keys, you can use a theta-join. Here is an example:

```
SELECT t FROM Topic t, Forum f WHERE t.postCount = f.forumPostCount
```

This query returns any **`Topic`** that has the same post count as a **`Forum`**. A theta-join allows you to join entities that otherwise might not have an explicit relationship or to join them on unrelated but similar information.

INNER JOIN

An inner join in JPQL is also referred to as a relationship join. The syntax of an inner join is:

```
[INNER] JOIN join_association_path_expression [AS] identification_variable
```

Both **`INNER`** and **`AS`** are optional; you can use them to more clearly document the intent of your query, although they do not affect the query in any way.

So what does **`join_assocation_path_expression`** mean? It means you are navigating an association-field of an entity — either a single-valued association or a collection. Figure 7.3 shows two inner join queries.

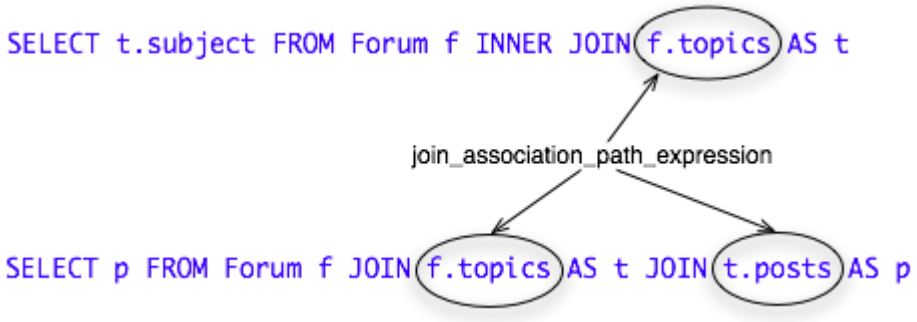

Figure 7.3: Two example join queries

Because you cannot navigate a collection association-field in a **SELECT** clause, the JPQL gives you the **INNER JOIN** operator. If you want to navigate the **Forum**-to-**Topic**-to-**Post** relationship (shown in Figure 7.2) and fetch all **Post titles**, you would create a query like this:

```
SELECT p.title FROM Forum f JOIN f.topics AS t JOIN t.posts AS p
```

This query would result in a list of zero or more **String** objects that represent the various post titles in which a **Forum** to **Topic** to **Post** join occurred.

LEFT OUTER JOIN

An outer join will return all instances of one entity and the instances of the other entity that match the join criteria. The syntax for a left join is

```
LEFT [OUTER] JOIN join_association_path_expression [AS] identification_variable
```

The **[OUTER]** reference is optional because the **LEFT JOIN** and **LEFT OUTER JOIN** operators are considered to be the same thing in JPQL. Using the **Forum**/**Post** entities in Figure 7.2, the following left join fetches all **Forum** instances and any **Topic** that is associated with a **Forum**. If no **Topic** is found, the second item in the **Object** array will be null:

```
SELECT f, t FROM Forum f LEFT JOIN f.topics t
```

Using my sample database, the above query returned the following results:

```
[ Object: [Forum] Object: [Topic] ]
[ Object: [Forum] Object: [null]  ]
[ Object: [Forum] Object: [null]  ]
[ Object: [Forum] Object: [null]  ]
[ Object: [Forum] Object: [null]  ]
```

The query returned all **Forum** instances and only one **Topic** because only one **Forum** has a **Topic** (looks like the message board needs some users).

Because a **LEFT JOIN** operator is an effective way to pre-fetch a relationship, the JPA created the **FETCH JOIN** operator. The **FETCH JOIN** operator is covered in the next section.

FETCH JOIN

A fetch join allows you to create queries that pre-fetch an otherwise lazy relationship. If you know a **LAZY** relationship will be needed after the entities have been fetched and the entities may be detached, you can pre-fetch the relationship using the **FETCH JOIN** operator. The **FETCH JOIN** syntax is:

```
[LEFT [OUTER] INNER] JOIN FETCH join_association_path_expression
```

Unlike the previous join definitions, **FETCH JOIN** does not have a range variable because you cannot use the implicitly referenced entity anywhere in the query. The following query will fetch any **Forum** entity that has a **Topic**. Only those **Forum** instances with **Topics** are fetched. Listing 7.4 shows how the **Forum** entity is defined. Note that the relationship to **Topic** is **Lazy**.

Listing 7.4

```
@Entity
public class Forum extends ModelBase implements Serializable {

    @OneToMany(fetch=FetchType.LAZY,
               cascade={CascadeType.PERSIST, CascadeType.MERGE},
               mappedBy="forum")
    @OrderBy("type asc, dateUpdated desc")
    Set<Topic> topics = new HashSet<Topic>();

    // ...
}
```

Here is the query:

```
SELECT DISTINCT f FROM Forum f JOIN FETCH f.topics
```

Because five forum instances exist in the database and only one has a topic, the above query will return one forum instance, and the topic's relationship will be eagerly fetched. If you did not use the **DISTINCT** operator, the persistence provider would retrieve one instance of forum for every topic in the system. By using the **DISTINCT** operator, any duplicate instances are removed.

To fetch all forum instances and eagerly fetch any topics they have, use **LEFT JOIN FETCH**, as in the following query:

```
SELECT DISTINCT f FROM Forum f LEFT JOIN FETCH f.topics
```

This query returns a list of unique forum instances, and the **topics** field is eagerly fetched. The only drawback to using the **JOIN FETCH** operator is the need to know the object model. Once you know the fetch type of the relationships, you can use **JOIN FETCH** to optimize your queries.

WHERE, GROUP BY, HAVING

The **WHERE** clause of a query is composed of conditional expressions that determine the entities to be retrieved. You can use the **GROUP BY** clause to aggregate the results of your query as long as the fields you **GROUP BY** appear in the **SELECT** clause. You can further refine your results with the **HAVING** operator. The JPA specification does not require persistence providers to support **HAVING** without **GROUP BY**; for portability, you might not want to use **HAVING** without **GROUP BY**.

Figure 7.4 is a diagram of a **Post**/**User** object.

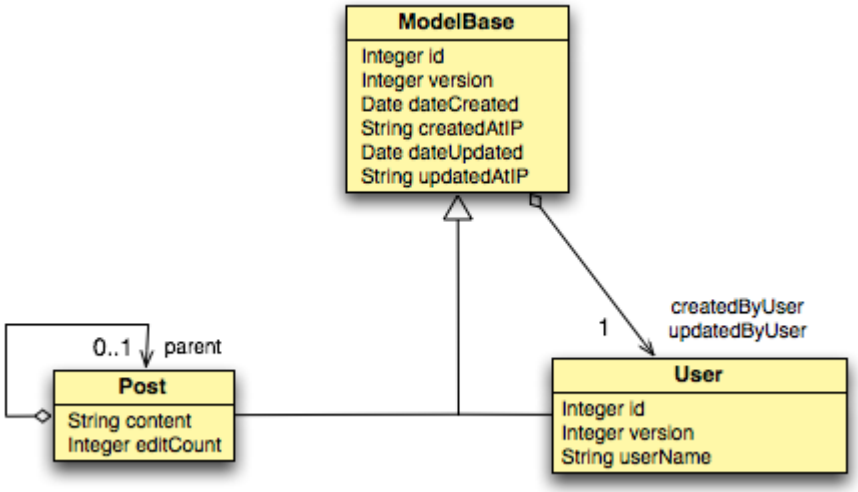

Figure 7.4: **Post**/**User** object diagram

Let's say you want to determine the number of posts each user in the **agoraBB** system has made. Using the objects in Figure 7.4, you will notice that no **User** to **Post** relationship exists. Because both **User** and **Post** inherit from the **ModelBase** **MappedSuperclass**, we know that each **Post** object has a **createdByUser** and **updatedByUser** field. Using the inner join syntax, you could write this query:

```
SELECT count(p) From Post p JOIN p.createdByUser u
```

The problem is that this query returns the total number of posts that have been created by a user using the **createdByUser** field. If you want to know how many posts each user has made, you need to use the **GROUP BY** operator:

```
SELECT u, count(p) From Post p JOIN p.createdByUser u GROUP BY u
```

This query returns a user entity and the number of posts the user has made. To further refine your results, you can filter the groups (identified by **GROUP BY**) with the **HAVING** operator. The following query returns users and their post count as long as they have a password with more than eight characters:

```
SELECT u, count(p) From Post p JOIN p.createdByUser u GROUP BY u HAVING length(u.password) > 8
```

CONDITIONAL EXPRESSIONS

Conditional expressions are used in the **WHERE** clause and **HAVING** clause of a JPQL query (**SELECT**, **UPDATE**, or **DELETE**). You should be aware of a couple of restrictions when you are creating a conditional expression:

- Including a **LOB** state-field in a conditional expression might not allow it to be portable across database vendors.
- String literals are enclosed in single quotes like `'this'`. If you need to use a single quote in your query, use two single quotes together. You cannot use a Java escape sequence in a query (for example, `\'` to represent a single quote). Boolean values are represented with the **TRUE** and **FALSE** literal (they are not case sensitive), numeric literals follow Java conversions, and date literals are not supported. **Enum** literals are also supported, but you must use the fully qualified name of the **Enum**, such as **com.sourcebeat.jpa.model.FTPType**.
- Identification variables must appear in the **FROM** clause of a **SELECT** or **DELTE** query. If you are writing an **UPDATE** query, the identification variable must appear in the **UPDATE** clause. Identification variables always represent the entity type for which they are defined and do not represent entities in a collection.
- You can use positional or named input parameters, but you cannot mix the two types within a given query. An input parameter can appear in the **WHERE** clause and/or the **HAVING** clause of a query.
  - The format for a positional parameter is question mark (**?**) followed by a positive integer starting from 1. For example, **?1**. You can repeat the same positional multiple times in a query, as in the following example:

    ```
    SELECT u FROM User u WHERE u.dateCreated = ?1 OR u.dateUpdated = ?1
    ```

  - A named parameter is represented by a colon (**:**) followed by a Java identifier, such as a java variable name. Named parameters look like this:

    ```
    SELECT u FROM User u WHERE u.dateCreated = :aDate OR u.dateUpdated = :aDate
    ```

FUNCTIONS AND EXPRESSIONS

JPQL supports functions, various **IN**, **LIKE**, and **BETWEEN** style expressions, and collection-oriented conditional expressions. This section details the various options that are available to you when you are writing a query.

The operator precedence in a JPQL query is:

- Navigation operator (.)
- Unary (+,-)
- Multiplication (*), Division (/)
- Addition (+), Subtraction (-)
- Comparison operators =, >, >=, <, <=,<> (not equal), [NOT] BETWEEN, [NOT] LIKE, [NOT] IN, IS [NOT] NULL, IS [NOT] EMPTY, [NOT] MEMBER [OF]
- Logical operators: NOT, AND, OR

BETWEEN

The **BETWEEN** comparison operator allows you to specify a range of values for the field of an entity. The syntax of **BETWEEN** is:

```
expression [NOT] BETWEEN expression AND expression
```

Where **expression** is a string, arithmetic, or datetime expression. Here are some example queries using the **BETWEEN** operator:

```
SELECT u FROM User u WHERE u.dateCreated between :startDate AND :endDate
SELECT t FROM Topic t WHERE t.postCount NOT BETWEEN ?1 AND ?2
```

IN

The **IN** comparison operator enables you to specify a list of values for a state-field. You can list one or more items as literals or parameter values (named or positional) or use a subquery to dynamically generate a list of values. **String**, numeric, or **enum** type state-fields can be used with the **IN** operator. The state-field type must match the type of the list of values. The syntax for the **IN** operator is:

```
state-field [NOT] in (item {, item2}* | subquery).
```

Here are a couple of examples:

```
SELECT f FROM Forum f WHERE f.type IN (?1, ?2)
SELECT f FROM Forum f WHERE f.type IN (1, 2)
```

LIKE

The **LIKE** operator allows you to search string fields for partial values. JPQL uses an underscore (_) to represent any one character in your search string. The percent character (%) represents a sequence of characters; all other characters represent themselves in the query. The general format of **LIKE** is:

```
string-expression [NOT] LIKE pattern [ESCAPE escape-char]
```

If you need to use the underscore or percent sign as a literal in your query, use the **ESCAPE** format. For example, use **forum.description like 'QA_%' ESCAPE '\'**. You must include the backslash before the underscore or percent sign, and you must include the **ESCAPE '\'** syntax after the search string. Here are some examples:

- '**tr_ck**' matches '**truck**' and '**trick**', but not '**trucker**'.
- '**tr%**' matches '**truck**', '**tractor**', '**trick**', and so on.
- '**tr_ck%**' matches '**truck**', '**trick**', and '**trucker**'.

If you want to search for the literal string '_hello', your query would be:

```
'\_hello' ESCAPE '\'
```

ESCAPE '\' tells the database: "I'm using the backslash as an escape character." Here is what this looks like in code:

```
em.createQuery("SELECT f FROM Forum f " +
    "WHERE f.description LIKE '\\_%' ESCAPE '\\'");
```

In this code, you use two backslash characters: The first for the Java compiler and the second for the JPQL parser.

If you try to execute the example **ESCAPE** query with the MySQL database, you might get a database exception. By default, MySQL is configured to recognize the backslash as an escape character, so telling it to treat the backslash as an escape character is an error. In order for your query to be portable across database vendors, you need to turn off backslash escapes in MySQL (available in version 5.0 and later) for all MySQL instances or in the JDBC connection. To disable backslash escapes for your JDBC connection, add the **sessionVariables** portion of the following URL to your JDBC URL:

```
jdbc:mysql://localhost:3306/db?sessionVariables=sql_mode=NO_BACKSLASH_ESCAPES
```

For more information, see the MySQL documentation.

IS NULL

The **IS NULL** comparison operator enables you to test for **NULL** fields — either a single-valued path expression or an input parameter. You can use **IS NOT NULL** to ensure a single-valued path expression has a value, or use **IS NULL** to check for **NULL** values.

```
SELECT p FROM PrivateMessage p WHERE p.dateRead IS NOT NULL
// toUser references a many-to-one relationship, so you can use IS [NOT] NULL
SELECT p FROM PrivateMessage p WHERE p.toUser IS NOT NULL
// this query does not work because we are using a
// collection-value path-expression
SELECT f FROM Forum f WHERE f.topics IS NULL
```

IS EMPTY

The **IS [NOT] EMPTY** operator is used to test for empty or not empty collection-value path-expressions.

```
// the above query rewritten to use IS EMPTY
SELECT f FROM Forum f WHERE f.topics IS EMPTY
// this query will find all forum entities with topics
// (i.e. the collection is not empty)
SELECT f FROM Forum f WHERE f.topics IS NOT EMPTY
```

MEMBER

The **[NOT] MEMBER [OF]** comparison operator allows you to determine whether an entity is part of a collection. The **[OF]** reference is optional and does not affect the **MEMBER** comparison operator. You can use it or leave it off.

Use **NOT MEMBER** to determine whether an entity is not part of a collection. The syntax of the **MEMBER** operator is:

```
Expression [NOT] MEMBER [OF] collection-valued path-expression
// find the forum instance that contains Topic t
Query q2 = em.createQuery("SELECT f FROM Forum f " +
    "WHERE :topic MEMBER f.topics");
q2.setParameter("topic", t);
List results2 = q2.getResultList();
```

STRING FUNCTIONS

The following string functions are supported as functional expressions in the **WHERE** or **HAVING** clause of JPQL query:

- **CONCAT(string 1, string 2)**: Appends string 2 to string 1.

- **SUBSTRING(string, starting position, length)**: Extracts **length** characters from **string** starting at **starting position**.
- **LOWER(string)**: Converts a **string** into lowercase.
- **UPPER(string)**: Converts a **string** into uppercase.
- **LENGTH(string)**: Returns the length of **string** as an integer.
- **TRIM([[LEADING|TRAILING|BOTH] [char] FROM] string)**:Trims all **leading**, **trailing**, or **both** instances of **char** from **string**. The simplest form of trim is **TRIM(string),** which removes **BOTH** the leading and trailing **space** char from **string**.
- **LOCATE(string1, string2 [,start])**: Returns the position of **string2** in **string1**. You can optionally define a **start** position for the locate function.

ARITHMETIC FUNCTIONS

The following arithmetic functions are supported as functional expressions in the **WHERE** or **HAVING** clause of a JPQL query:

- **ABS(arithmetic expression)**: Returns the unsigned value of arithmetic expressions.
- **SQRT(arithmetic expression)**: Returns the square root of **arithmetic expression** as a **Double**.
- **MOD(arithmetic expression 1, arithmetic expression 2)**: Takes the modulus for argument one and two and returns an integer.
- **SIZE(collection-valued path-expression)**: Calculates the number of elements in a collection and returns an integer. If the collection is empty, **SIZE** returns zero (0).

DATETIME FUNCTIONS

The following **Datetime** functions are supported:

- **CURRENT_DATE**: The current date, as defined by the database.
- **CURRENT_TIME**: The current time, as defined by the database.
- **CURRENT_TIMESTAMP**: The current date and time, as defined by the database.

THE SELECT FUNCTION

The **SELECT** clause identifies the result of the query. The **SELECT** clause contains one or more of the following elements:

- A path expression or identification variable: Identifies an entity to be returned.
- A single-valued path-expression: Specifies a field or an entity to be returned.
- An aggregate **SELECT** expression: Indicates a calculated value is returned (for example, **COUNT (e)**)
- A constructor expression: Allows you to return a new object from the items selected.

The **SELECT** clause enables you to query for a large variety of entities, calculated values, projected values, and non-entity classes. You cannot use a collection-value path-expression in a **SELECT** clause; therefore, the following query is not valid:

```
SELECT f.topics FROM Forum f
```

As noted earlier, some of the current JPA persistence providers allow this type of query. For portability, you should use the following query instead (see the *Joins* section):

```
SELECT t FROM Forum f JOIN f.topics t
```

The result of your query can be an abstract schema type (entity), a state-field (field or property of an entity), the result of an aggregate function, an object created with the **NEW** operator, or any combination of these possibilities. If you select an abstract schema type or construct a new object, the result of your query will be a list of objects with types that are the entity or new object. If you use aggregate functions, select state-fields, or select a variety of types, the result of your query will be a list of object array (**Object[]**) instances. The objects in the array correspond to the order in which they were specified in the query. For example:

```
SELECT t.subject, t.content FROM  Topic t
```

The query returns a list of **Object[]** instances. Each item in the list consists of two **String** objects: The first item (**index 0**) is the subject, and the second item (**index 1**) is the content of the **Topic**.

CONSTRUCTOR EXPRESSION

You can create a new object as the result of your query. The object does not need to be an entity, but it does need a constructor that matches the order and type of your **SELECT** clause. Listing 7.5 is a transient object to hold user statistics:

Listing 7.5

```
public class UserStatistics {

    private String username;
    private Integer userId;
    private long postCount;

    public UserStatistics(String username, Integer userId, long postCount) {
        super();
        this.username = username;
        this.userId = userId;
        this.postCount = postCount;
    }
    // getter methods removed for readability
}
```

The following query will calculate the number of posts each user has made in the system and store the result, the user name, and the user ID in a **`UserStatistics`** transient object:

```
Query q = em.createQuery("SELECT NEW com.sourcebeat.jpa.model.UserStatistics("
    +"u.username, u.id, COUNT(p)) "
    + "FROM Post p JOIN p.createdByUser u "
    + "WHERE p.parent IS NOT NULL GROUP BY u");
List results = q.getResultList();
```

AGGREGATE FUNCTIONS

Use the following aggregate functions (applied to a path expression) in a **`SELECT`** clause.

AVG

Calculates the average value of the numeric argument across the result of the query and returns a **`Double`** integer

```
SELECT AVG(f.forumPostCount) FROM Forum f
```

COUNT

Calculates the total number of entities found and returns a **`Long`** integer. If no entities are found, **`COUNT`** returns **`0`**

```
SELECT COUNT(f) FROM Forum f
```

MAX

Calculates the maximum value of the argument across the result of the query and returns the same types as the argument. The **MAX** function can be applied to any orderable state-field, including numeric types, strings, character types, or dates.

```
SELECT MAX(f.forumPostCount) FROM Forum f
```

MIN

Calculates the minimum value of the argument across the result of the query and returns the same types as the argument. The **MIN** function can be applied to any orderable state-field, including numeric types, strings, character types, or dates.

```
SELECT MIN(f.dateCreated) FROM Forum f
```

SUM

Calculates the sum of the numeric argument across the result of the query and returns a **Double** integer when the argument is a floating point type, a **BigInteger** when a **BigInteger** type is used, and a **BigDecimal** integer when the argument is a **BigDecimal**.

```
SELECT SUM(f.forumPostCount) FROM Forum f
```

RULES OF USAGE

These functions — except for **COUNT** — must be applied to a path expression that ends in a state-field. You can use a state-field, an association-field, or an identification variable as an argument to the **COUNT** function.

If no value exists for the **SUM**, **AVG**, **MIN**, and **MAX** functions, they return **NULL**.

To remove duplicates from your query before the aggregate function is applied, use the **DISTINCT** operator; however, it is illegal to use the **DISTINCT** operator with **MAX** or **MIN**. In addition, **NULL** values are removed before the functions calculate their results, regardless of whether **DISTINCT** is used.

The return type of these functions is important when you are using a constructor-expression **SELECT** statement. The **postCount** property of the **UserAverages** object is a **Long**. Initially, I made it an **int**, but when the query ran, Hibernate threw an **IllegalArgumentException** indicating no appropriate constructor was present in **UserStatistics**. Once I realized that **COUNT** returns a **Long**, I updated **UserStatistics**, and the query worked correctly.

ORDER BY

The **ORDER BY** operator allows you to order the results of your query. The database will do a more efficient job of sorting than your application will. To maintain the order of your collection, use the **ORDER BY** operator (see the **@OrderBy** annotation in *Chapter 3: Entities Part II* for more information).

Here is where the **ORDER BY** operator is placed in a JPQL query:

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

The syntax for **ORDER BY** is:

```
ORDER BY expression [ASC | DESC] {, expression [ASC | DESC]}*
```

Here are a few valid **ORDER BY** examples:

- **SELECT u FROM User u ORDER BY u**
- **SELECT u FROM User u ORDER BY u.address**
- **SELECT u.username, u.id FROM User u ORDER BY u.username DESC**

expression can be an identification variable, a single-value association path, or a state-field path expression; the queries here illustrate each of these types of **ORDER BY**.

You should be aware of a couple of restrictions for using the **ORDER BY** operator:

- When using an identification variable or single-valued association path, the item you are ordering your query by must be an orderable type, such as a numeric type, a string, a character type, or a date.
- If you use a state-field path expression, it must also appear in the **SELECT** clause.

ASC represents ascending order (smallest to largest) and is the default ordering. **DESC** represents descending order (largest to smallest) and is used only if you explicitly add it to the **ORDER BY** clause of your query. Sorting precedence is from left to right in the order you list the fields.

With these conditions in mind, the following query is not valid because it orders on a field that is not included in the **SELECT** clause:

```
SELECT u.username, u.id FROM User u ORDER BY u.password
```

BULK OPERATIONS

The JPQL provides an alternative for updating and deleting one or more entities in a single statement. The bulk operation support in JPQL works with one entity type (and its subclasses) at a time; that is, you can identify only one entity in the **FROM** or **UPDATE** clause. Here is the syntax for an **UPDATE** query:

```
UPDATE abstract_persistence_scheama_name [AS] identification_variable
    SET state_field | single_value_association_field = value
      {,state_field | single_value_association_field = value }*
```

The **value** reference must be compatible with the type of the state-field or single-value association-field you are updating. You can use any of the following values for **value**:

- an arithmetic expression
- string
- **datetime**
- **boolean**
- **enum**
- simple entity expression
- **NULL**

The **DELETE** query looks like this:

```
DELETE FROM abstract_persistence_scheama_name [[AS] identification_variable]
    [WHERE clause]
```

The syntax of the **WHERE** clause is the same as a **SELECT** query. (See the **WHERE clause** section for more information.) The **DELETE** operation affects only the entity and the subclasses that are identified in the **FROM** clause. The operation is not cascaded to any related entities. In addition, the **UPDATE** operation does not update the entity's version column.

Bulk operations are converted into SQL and executed in the database, bypassing the persistence context. When using a transaction-scoped persistence context, execute bulk operations in their own transaction or at the beginning of a transaction.

Bulk operations and extended persistence contexts are a difficult combination to manage. Because an extended persistence context isn't synchronized with the database until it joins a transaction, you might have entities that were deleted but are still in the persistence-context.

Persistence providers typically invalidate some of their cache before you execute bulk operations. Depending on the provider, some or all of the cache may be invalidated. Using bulk operations frequently can impact the performance of your application. For these reasons, you should perform bulk operations in their own transactions or at the beginning of a transaction.

EXAMPLES

When you create an **UPDATE** or **DELETE** query, you need to use the Query API method, **executeUpdate(),** to perform the update or delete. If you use **getResultList()** or **getSingleResult()**, the persistence provider throws an **IllegalStateException**. Also, if you try to execute a **SELECT** query with the **executeUpdate()** method, the persistence provider throws an **IllegalStateException**.

The following listings are some bulk **UPDATE** examples:

- Double the post count of a forum. See Listing 7.6:

Listing 7.6

```
Query q2 = em.createQuery("UPDATE Forum AS f "
    +"SET f.forumPostCount = f.forumPostCount * 2");

q2.executeUpdate();
```

- Set the **dateUpdated** field of all **Role** entities to the current date and time. See Listing 7.7:

Listing 7.7

```
Query q = em.createQuery("UPDATE Role AS r " +
    "SET r.dateUpdated = CURRENT_TIMESTAMP");

q.executeUpdate();
```

- Set the Boolean field (**pruningEnabled**) to **true**. In the **agoraBB** application, an **EntityListener** class normally sets **dateUpdated** and **updatedByUser** fields. The persistence provider manages the **version** column. Because these hooks are bypassed when you perform a bulk operation, the query is written to

update these fields. Also, note that **executeUpdate** will return the number of entities updated (or deleted when using **DELETE**). See Listing 7.8:

Listing 7.8

```
// Assume we already fetched the correct User identified by
// the variable adminUser
Query forumUpdate = em.createQuery("UPDATE Forum AS f " +
    "SET f.pruningEnabled = TRUE, f.dateUpdated = CURRENT_TIMESTAMP, " +
    "f.version = f.version + 1, f.updatedByUser = :user");

forumUpdate.setParameter("user", adminUser);

int entitiesUpdated = forumUpdate.executeUpdate();
```

You can reset the change to the **pruningEnabled** field in Listing 7.8 by setting the field to **null**, as shown in Listing 7.9:

Listing 7.9

```
Query pruningReset = em.createQuery("UPDATE Forum AS f " +
    "SET f.pruningEnabled = NULL");

pruningReset.executeUpdate();
```

This update sets the value of an **enum** field. You need to use the fully qualified name of the **enum** class because the **enum** is not an entity and the persistence provider does not know anything about **Status**, although it can resolve **com.sourcebeat.jpa.model.Status**. See Listing 7.10:

Listing 7.10

```
Query enumUpdate = em.createQuery("UPDATE Forum AS f " +
    "SET f.status = com.sourcebeat.jpa.model.Status.LOCKED " +
    "WHERE f.type = com.sourcebeat.jpa.model.FTPType.ANNONUCEMENT");

int enumUpdateCount = enumUpdate.executeUpdate();
```

Here are a couple of **DELETE** examples

- Delete all **User**s from the system who do not have a password. See Listing 7.11.

Listing 7.11

```
Query removeRoles = em.createQuery("DELETE FROM User u " +
    "WHERE u.password = NULL");

int rolesRemoved = removeRoles.executeUpdate();
```

- Delete **`Forum`** instances that do not have any **`Topics`**. See Listing 7.12:

Listing 7.12

```
Query removeForums = em.createQuery("DELETE FROM Forum f " +
    "WHERE f.topics IS EMPTY");

int forumsRemoved = removeForums.executeUpdate();
```

SUMMARY

The JPQL provides a variety of options for selecting, grouping, ordering, and summarizing your entities. It includes rich support for joins and the ability to eagerly fetch lazy relationships. You have explored the various functions and expressions supported and worked through all the options available to you with the **`SELECT`** clause. The final section of this chapter ends with bulk operations, which is the ability to affect one or more entities with a single query (actually a **`DELETE`** or **`UPDATE`** statement).

You should now be familiar enough with the options provided to you by the JPQL to create your own query from the simple to the complex.