# Pro EJB 3

## Java Persistence API

Mike Keith

Merrick Schincariol

Apress®

**Pro EJB 3: Java Persistence API**

**Copyright © 2006 by Mike Keith and Merrick Schincariol**

ISBN-13 (pbk): 978-1-59059-645-6

ISBN-10 (pbk): 1-59059-645-5

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Steve Anglin
Technical Reviewer: Jason Haley, Huyen Nguyen, Shahid Shah
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Gennick, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Keir Thomas, Matt Wade
Project Manager: Julie M. Smith
Copy Edit Manager: Nicole LeClerc
Copy Editor: Hastings Hart
Assistant Production Director: Kari Brooks-Copony
Production Editor: Laura Esterman
Compositors: Pat Christenson and Susan Glinert Stevens
Proofreader: Elizabeth Berry
Indexer: Julie Grady
Artist: Kinetic Publishing Services, LLC
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Source Code section. You will need to answer questions pertaining to this book in order to successfully download the code.

CHAPTER 6

# Using Queries

For most enterprise applications, getting data out of the database is at least as important as the ability to put new data in. From searching, to sorting, to analytics and business intelligence, efficiently moving data from the database to the application and presenting it to the user is a regular part of enterprise development. Doing so requires the ability to issue bulk queries against the database and interpret the results for the application. Although high-level languages and expression frameworks have attempted to insulate developers from the task of dealing with database queries at a low level, it's probably fair to say that most enterprise developers have worked with at least one SQL dialect at some point in their career.

Object-relational mapping adds another level of complexity to this task. Most of the time, the developer will want the results converted to entities so that the query results may be used directly by application logic. Similarly, if the domain model has been abstracted from the physical model via object-relational mapping, then it makes sense to also abstract queries away from SQL, which is not only tied to the physical model but also difficult to port between vendors. Fortunately, as we will see, the Java Persistence API can handle a diverse set of query requirements.

The Java Persistence API supports two query languages for retrieving entities and other persistent data from the database. The primary language is Java Persistence QL (JPQL), a database-independent query language that operates on the logical entity model as opposed to the physical data model. Queries may also be expressed in SQL in order to take advantage of the underlying database. We will discuss SQL queries with the Java Persistence API later in Chapter 9.

We will begin our discussion of queries with an introduction to Java Persistence QL, followed by an exploration of the query facilities provided by the `EntityManager` and `Query` interfaces.

## Java Persistence QL

Before discussing JPQL, we must first look to its roots in the EJB specification. The EJB Query Language (EJB QL) was first introduced in the EJB 2.0 specification to allow developers to write portable finder and select methods for container-managed entity beans. Based on a small subset of SQL, it introduced a way to navigate across entity relationships both to select data and to filter the results. Unfortunately it placed strict limitations on the structure of the query, limiting results to either a single entity or a persistent field from an entity. Inner joins between entities were possible but used an odd notation. The initial release didn't even support sorting.

The EJB 2.1 specification tweaked EJB QL a little bit, adding support for sorting, and introduced basic aggregate functions; but again the limitation of a single result type hampered the use of aggregates. You could filter the data, but there was no equivalent to SQL GROUP BY and HAVING expressions.

Java Persistence Query Language significantly extends EJB QL, eliminating many weaknesses of the previous versions while preserving backwards compatibility. The following features are available above and beyond EJB QL:

- Single and multiple value result types
- Aggregate functions, with sorting and grouping clauses
- A more natural join syntax, including support for both inner and outer joins
- Conditional expressions involving subqueries
- Update and delete queries for bulk data changes
- Result projection into non-persistent classes

The next few sections provide a quick introduction to Java Persistence QL intended for readers familiar with SQL or previous versions of EJB QL. A complete tutorial and reference for Java Persistence QL can be found in Chapter 7.

## Getting Started

The simplest JPQL query selects all of the instances of a single entity type. Consider the following query:

```
SELECT e
FROM Employee e
```

If this looks similar to SQL, it should. JPQL uses SQL syntax where possible in order to give developers experienced with SQL a head start in writing queries. The key difference between SQL and JPQL for this query is that instead of selecting from a table, an entity from the application domain model has been specified instead. The SELECT clause of the query is also slightly different, listing only the `Employee` alias `e`. This indicates that the result type of the query is the `Employee` entity, so executing this statement will result in a list of zero or more `Employee` instances.

Starting with an alias, we can navigate across entity relationships using the dot (`.`) operator. For example, if we want just the names of the employees, the following query will suffice:

```
SELECT e.name
FROM Employee e
```

Each part of the expression corresponds to a persistent field of the entity or an association leading to another entity or collection of entities. Since the `Employee` entity has a persistent field named `name` of type `String`, this query will result in a list of zero or more `String` objects.

We can also select an entity we didn't even list in the FROM clause. Consider the following example:

```
SELECT e.department
FROM Employee e
```

An employee has a many-to-one relationship with his or her department named `department`, so therefore the result type of the query is the `Department` entity.

## Filtering Results

Just like SQL, JPQL supports the WHERE clause to set conditions on the data being returned. The majority of operators commonly available in SQL are available in JPQL, including basic comparison operators; IN, LIKE, and BETWEEN expressions; numerous function expressions (such as SUBSTRING and LENGTH); and subqueries. The key difference for JPQL is that entity expressions and not column references are used. Listing 6-1 demonstrates filtering using entity expressions in the WHERE clause.

**Listing 6-1.** *Filtering Criteria Using Entity Expressions*

```
SELECT e
FROM Employee e
WHERE e.department.name = 'NA42' AND
      e.address.state IN ('NY','CA')
```

## Projecting Results

For applications that need to produce reports, a common scenario is selecting large numbers of entity instances, but only using a portion of that data. Depending on how an entity is mapped to the database, this can be an expensive operation if much of the entity data is discarded. It would be useful to return only a subset of the properties from an entity. The following query demonstrates selecting only the name and salary of each `Employee` instance:

```
SELECT e.name, e.salary
FROM Employee e
```

## Joins Between Entities

The result type of a select query cannot be a collection; it must be a single valued object such as an entity instance or persistent field type. This means that expressions such as `e.phones` are illegal in the SELECT clause because they would result in `Collection` instances. Therefore, just as with SQL and tables, if we want to navigate along a collection association and return elements of that collection, then we must join the two entities together. Listing 6-2 demonstrates a join between `Employee` and `Phone` entities in order to retrieve all of the cell phone numbers for a specific department.

**Listing 6-2.** *Joining Two Entities Together*

```
SELECT p.number
FROM Employee e, Phone p
WHERE e = p.employee AND
      e.department.name = 'NA42' AND
      p.type = 'Cell'
```

In JPQL, joins may also be expressed in the FROM clause using the JOIN operator. The advantage of this operator is that the join can be expressed in terms of the association itself, and the query engine will automatically supply the necessary join criteria when it generates the SQL. Listing 6-3 shows the same query rewritten to use the JOIN operator. Just as in the previous query, the alias `p` is of type `Phone`, only this time it refers to each of the phones in the `e.phones` collection.

**Listing 6-3.** *Joining Two Entities Together Using the JOIN Operator*

```
SELECT p.number
FROM Employee e JOIN e.phones p
WHERE e.department.name = 'NA42' AND
      p.type = 'Cell'
```

JPQL supports multiple join types, including inner and outer joins, as well as a technique called fetch joins for eagerly loading data associated to the result type of a query but not directly returned. See the Joins section in Chapter 7 for more information.

## Aggregate Queries

The syntax for aggregate queries in JPQL is very similar to that of SQL. There are five supported aggregate functions (AVG, COUNT, MIN, MAX, and SUM), and results may be grouped in the GROUP BY clause and filtered using the HAVING clause. Once again, the difference is the use of entity expressions when specifying the data to be aggregated. Listing 6-4 demonstrates an aggregate query with JPQL.

**Listing 6-4.** *Query Returning Statistics for Departments with Five or More Employees*

```
SELECT d, COUNT(e), MAX(e.salary), AVG(e.salary)
FROM Department d JOIN d.employees e
GROUP BY d
HAVING COUNT(e) >= 5
```

## Query Parameters

JPQL supports two types of parameter binding syntax. The first is positional binding, where parameters are indicated in the query string by a question mark followed by the parameter number. When the query is executed, the developer specifies the parameter number that should be replaced. Listing 6-5 demonstrates positional parameter syntax.

**Listing 6-5.** *Positional Parameter Notation*

```
SELECT e
FROM Employee e
WHERE e.department = ?1 AND
      e.salary > ?2
```

Named parameters may also be used and are indicated in the query string by a colon followed by the parameter name. When the query is executed, the developer specifies the parameter name that should be replaced. Listing 6-6 demonstrates named parameter syntax.

**Listing 6-6.** *Named Parameter Notation*

```
SELECT e
FROM Employee e
WHERE e.department = :dept AND
      e.salary > :base
```

## Defining Queries

The Java Persistence API provides the `Query` interface to configure and execute queries. An implementation of the `Query` interface for a given query is obtained through one of the factory methods in the `EntityManager` interface. The choice of factory method depends on the type of query (JPQL or SQL) and whether or not the query has been predefined. For now, we will restrict our discussion to JPQL queries. SQL query definition is discussed in Chapter 9.

There are two approaches to defining a query. A query may either be dynamically specified at runtime or configured in persistence unit metadata (annotation or XML) and referenced by name. Dynamic queries are nothing more than strings, and therefore may be defined on the fly as the need arises. Named queries, on the other hand, are static and unchangeable but are more efficient to execute as the persistence provider can translate the JPQL string to SQL once when the application starts as opposed to every time the query is executed.

The following sections compare the two approaches and discuss when one should be used instead of the other.

### Dynamic Query Definition

A query may be defined dynamically by passing the JPQL query string to the `createQuery()` method of the `EntityManager` interface. There are no restrictions on the query definition. All JPQL query types are supported, as well as the use of parameters. The ability to build up a string at runtime and use it for a query definition is useful, particularly for applications where the user may specify complex criteria and the exact shape of the query cannot be known ahead of time.

An issue to consider with dynamic queries, however, is the cost of translating the JPQL string to SQL for execution. A typical query engine will have to parse the JPQL string into a syntax tree, get the object-relational mapping metadata for each entity in each expression, and then generate the equivalent SQL. For applications that issue many queries, the performance cost of dynamic query processing can become an issue.

Many query engines will cache the translated SQL for later use, but this can easily be defeated if the application does not use parameter binding and concatenates parameter values directly into query strings. This has the effect of generating a new and unique query every time a query that requires parameters is constructed.

Consider the session bean method shown in Listing 6-7 that searches for salary information given the name of a department and the name of an employee. There are two problems with this example, one performance-related and one security-related. Because the names are concatenated into the string instead of using parameter binding, it is effectively creating a new and unique query each time. One hundred calls to this method could potentially generate one hundred different query strings. This not only requires excessive parsing of JPQL but also almost certainly makes it difficult for the persistence provider if it attempts to build a cache of converted queries.

**Listing 6-7.** *Defining a Query Dynamically*

```
@Stateless
public class QueryServiceBean implements QueryService {
    @PersistenceContext(unitName="DynamicQueries")
    EntityManager em;

    public long queryEmpSalary(String deptName, String empName) {
        String query = "SELECT e.salary " +
                       "FROM Employee e " +
                       "WHERE e.department.name = '" + deptName + "' AND " +
                       "      e.name = '" + empName + "'";
        return (Long) em.createQuery(query).getSingleResult();
    }
}
```

The second problem with this example is that a malicious user could pass in a value that alters the query to his advantage. Consider a case where the department argument was fixed by the application but the user was able to specify the employee name (the manager of the department is querying the salaries of his or her employees, for example). If the name argument were actually the text "`'_UNKNOWN' OR e.name = 'Roberts'`". The actual query parsed by the query engine would be as follows:

```
SELECT e.salary
FROM Employee e
WHERE e.department.name = 'NA65' AND
      e.name = '_UNKNOWN' OR
      e.name = 'Roberts'
```

By introducing the OR condition, the user has effectively given himself access to the salary value for any employee in the company, since the original AND condition has a higher precedence than OR and the fake employee name is unlikely to belong to a real employee in that department.

Parameter binding defeats this type of security threat, because the original query string is never altered. The parameters are marshaled using the JDBC API and handled directly by the database. The text of a parameter string is effectively quoted by the database, so the malicious attack would actually end up producing the following query:

```
SELECT e.salary
FROM Employee e
WHERE e.department.name = 'NA65' AND
      e.name = '_UNKNOWN'' OR e.name = ''Roberts'
```

The single quotes used in the query parameter here have been escaped by prefixing them with an additional single quote. This removes any special meaning from them, and the entire sequence is treated as a single string value.

This type of problem may sound unlikely, but in practice many web applications take text submitted over a GET or POST request and blindly construct queries of this sort without considering side effects. One or two attempts that result in a parser stack trace displayed to the web page and the attacker will learn everything he needs to know about how to alter the query to his advantage.

Listing 6-8 shows the same method as in Listing 6-7 except that it uses named parameters instead. This not only reduces the number of unique queries parsed by the query engine, but it also eliminates the chance of the query being altered.

**Listing 6-8.** *Using Parameters with a Dynamic Query*

```
@Stateless
public class QueryServiceBean implements QueryService {
    private static final String QUERY =
        "SELECT e.salary " +
        "FROM Employee e " +
        "WHERE e.department.name = :deptName AND " +
        "      e.name = :empName ";

    @PersistenceContext(unitName="DynamicQueries")
    EntityManager em;

    public long queryEmpSalary(String deptName, String empName) {
        return (Long) em.createQuery(QUERY)
                        .setParameter("deptName", deptName)
                        .setParameter("empName", empName)
                        .getSingleResult();
    }
}
```

We recommend statically defined named queries in general, particularly for queries that are executed frequently. If dynamic queries are a necessity, take care to use parameter binding instead of concatenating parameter values into query strings in order to minimize the number of distinct query strings parsed by the query engine.

## Named Query Definition

Named queries are a powerful tool for organizing query definitions and improving application performance. A named query is defined using the @NamedQuery annotation, which may be placed on the class definition for any entity. The annotation defines the name of the query, as well as the query text. Listing 6-9 shows how the query string used in Listing 6-8 would be declared as a named query.

**Listing 6-9.** *Defining a Named Query*

```
@NamedQuery(name="findSalaryForNameAndDepartment",
            query="SELECT e.salary " +
                  "FROM Employee e " +
                  "WHERE e.department.name = :deptName AND " +
                  "      e.name = :empName")
```

Note the use of string concatenation in the annotation definition. Formatting your queries visually aids in the readability of the query definition. Named queries are typically placed on the entity class that most directly corresponds to the query result, so the Employee entity would be a good location for this named query.

The name of the query is scoped to the persistence unit and must be unique within that scope. This is an important restriction to keep in mind, as commonly used query names such as “findAll” will have to be qualified for each entity. A common practice is to prefix the query name with the entity name. For example, the “findAll” query for the Employee entity would be named “Employee.findAll”. It is undefined what should happen if two queries in the same persistence unit have the same name, but it is likely that either deployment of the application will fail or one will overwrite the other, leading to unpredictable results at runtime. Entity-scoped query names are planned for the next release of the Java Persistence API and will remove the need for this kind of prefixing.

If more than one named query is to be defined for a class, they must be placed inside of a @NamedQueries annotation, which accepts an array of one or more @NamedQuery annotations. Listing 6-10 shows the definition of several queries related to the Employee entity. Queries may also be defined (or redefined) using XML. This technique is discussed in Chapter 10.

**Listing 6-10.** *Multiple Named Queries for an Entity*

```
@NamedQueries({
    @NamedQuery(name="Employee.findAll",
                query="SELECT e FROM Employee e"),
    @NamedQuery(name="Employee.findByPrimaryKey",
                query="SELECT e FROM Employee e WHERE e.id = :id"),
    @NamedQuery(name="Employee.findByName",
                query="SELECT e FROM Employee e WHERE e.name = :name")
})
```

Because the query string is defined in the annotation, it cannot be altered by the application at runtime. This contributes to the performance of the application and helps to prevent the kind of security issues we discussed in the previous section. Due to the static nature of the query string, any additional criteria that are required for the query must be specified using query parameters. Listing 6-11 demonstrates using the `createNamedQuery()` call on the `EntityManager` interface to create and execute a named query that requires a query parameter.

**Listing 6-11.** *Executing a Named Query*

```
@Stateless
public class EmployeeServiceBean implements EmployeeService {
    @PersistenceContext(unitName="EmployeeService")
    EntityManager em;

    public Employee findEmployeeByName(String name) {
         return (Employee) em.createNamedQuery("Employee.findByName")
                             .setParameter("name", name)
                             .getSingleResult();
    }

    // ...
}
```

Named parameters are the most practical choice for named queries as it effectively self-documents the application code that invokes the queries. Positional parameters are still supported, however, and may be used instead.

## Parameter Types

As mentioned earlier, the Java Persistence API supports both named and positional parameters for JPQL queries. The query factory methods of the entity manager return an implementation of the `Query` interface. Parameter values are then set on this object using the `setParameter()` methods of the `Query` interface.

There are three variations of this method for both named parameters and positional parameters. The first argument is always the parameter name or number. The second argument is the object to be bound to the named parameter. `Date` and `Calendar` parameters also require a third argument that specifies whether the type passed to JDBC is a `java.sql.Date`, `java.sql.Time`, or `java.sql.TimeStamp` value.

Consider the following named query definition, which requires two named parameters:

```
@NamedQuery(name="findEmployeesAboveSal",
            query="SELECT e " +
                  "FROM Employee e " +
                  "WHERE e.department = :dept AND " +
                  "      e.salary > :sal")
```

This query highlights one of the nice features of JPQL in that entity types may be used as parameters. When the query is translated to SQL, the necessary primary key columns will be inserted into the conditional expression and paired with the primary key values from the parameter. It is not necessary to know how the primary key is mapped in order to write the query. Binding the parameters for this query is a simple case of passing in the required `Department` entity instance as well as a `long` representing the minimum salary value for the query. Listing 6-12 demonstrates how to bind the entity and primitive parameters required by this query.

**Listing 6-12.** *Binding Named Parameters*

```
@Stateless
public class EmployeeServiceBean implements EmployeeService {
    @PersistenceContext(unitName="EmployeeService")
    EntityManager em;

    public List findEmployeesAboveSal(Department dept, long minSal) {
         return em.createNamedQuery("findEmployeesAboveSal")
                  .setParameter("dept", dept)
                  .setParameter("sal", minSal)
                  .getResultList();
    }

    // ...
}
```

Date and Calendar parameters are a special case because they represent both dates and times. In Chapter 4, we discussed mapping temporal types by using the @Temporal annotation and the TemporalType enumeration. This enumeration indicates whether the persistent field is a date, time, or timestamp. When a query uses a Date or Calendar parameter, it must select the appropriate temporal type for the parameter. Listing 6-13 demonstrates binding parameters where the value should be treated as a date.

**Listing 6-13.** *Binding Date Parameters*

```
@Stateless
public class EmployeeServiceBean implements EmployeeService {
    @PersistenceContext(unitName="EmployeeService")
    EntityManager em;

    public List findEmployeesHiredDuringPeriod(Date start, Date end) {
         return em.createQuery("SELECT e " +
                               "FROM Employee e " +
                               "WHERE e.startDate BETWEEN ?1 AND ?2")
                  .setParameter(1, start, TemporalType.DATE)
                  .setParameter(2, end, TemporalType.DATE)
                  .getResultList();
    }

    // ...
}
```

One thing to keep in mind with query parameters is that the same parameter can be used multiple times in the query string yet only needs to be bound once using the setParameter()

method. For example, consider the following named query definition, where the "dept" parameter is used twice in the WHERE clause:

```
@NamedQuery(name="findHighestPaidByDepartment",
            query="SELECT e " +
                  "FROM Employee e " +
                  "WHERE e.department = :dept AND " +
                  "      e.salary = (SELECT MAX(e.salary) " +
                  "                  FROM Employee e " +
                  "                  WHERE e.department = :dept)")
```

To execute this query, the "dept" parameter only needs to be set once with `setParameter()` as in the following example:

```
public Employee findHighestPaidByDepartment(Department dept) {
    return (Employee) em.createNamedQuery("findHighestPaidByDepartment")
                        .setParameter("dept", dept)
                        .getSingleResult();
}
```

## Executing Queries

The `Query` interface provides three different ways to execute a query, depending on whether or not the query returns results and how many results should be expected. For queries that return values, the developer may choose to call either `getSingleResult()` if the query is expected to return a single result or `getResultList()` if more than one result may be returned. The `executeUpdate()` method of the query interface is used to invoke bulk update and delete queries. We will discuss this method later in the section Bulk Update and Delete.

The simplest form of query execution is via the `getResultList()` method. It returns a collection containing the query results. If the query did not return any data, then the collection is empty. The return type is specified as a `List` instead of `Collection` in order to support queries that specify a sort order. If the query uses the ORDER BY clause to specify a sort order, then the results will be put into the result list in the same order. Listing 6-14 demonstrates how a query might be used to generate a menu for a command line application that displays the name of each employee working on a project as well as the name of the department that the employee is assigned to. The results are sorted by the name of the employee. Queries are unordered by default.

**Listing 6-14.** *Iterating over Sorted Results*

```
public void displayProjectEmployees(String projectName) {
    List result = em.createQuery("SELECT e " +
                                 "FROM Project p JOIN p.employees e " +
                                 "WHERE p.name = ?1 " +
                                 "ORDER BY e.name")
                    .setParameter(1, projectName)
                    .getResultList();
    int count = 0;
    for (Iterator i = result.iterator(); i.hasNext();) {
        Employee e = (Employee) i.next();
        System.out.println(++count + ": " + e.getName() + ", " +
                           e.getDepartment().getName());
    }
}
```

The `getSingleResult()` method is provided as a convenience for queries that return only a single value. Instead of iterating to the first result in a collection, the object is directly returned. It is important to note, however, that `getSingleResult()` behaves differently than `getResultList()` in how it handles unexpected results. Whereas `getResultList()` returns an empty collection when no results are available, `getSingleResult()` throws a `NoResultException` exception. Therefore if there is a chance that the desired result may not be found, then this exception needs to be handled.

If multiple results are available after executing the query instead of the single expected result, `getSingleResult()` will throw a `NonUniqueResultException` exception. Again, this can be problematic for application code if the query criteria may result in more than one row being returned in certain circumstances. Although `getSingleResult()` is convenient to use, be sure that the query and its possible results are well understood, otherwise application code may have to deal with an unexpected runtime exception. Unlike other exceptions thrown by entity manager operations, these exceptions will not cause the provider to roll back the current transaction, if there is one.

`Query` objects may be reused as often as needed so long as the same persistence context that was used to create the query is still active. For transaction-scoped entity managers, this limits the lifetime of the `Query` object to the life of the transaction. Other entity manager types may reuse `Query` objects until the entity manager is closed or removed.

Listing 6-15 demonstrates caching a `Query` object instance on the bean class of a stateful session bean that uses an extended persistence context. Whenever the bean needs to find the list of employees who are currently not assigned to any project, it reuses the same `unassignedQuery` object that was initialized during PostConstruct.

**Listing 6-15.** *Reusing a Query Object*

```
@Stateful
public class ProjectManagerBean implements ProjectManager {
    @PersistenceContext(unitName="EmployeeService",
                        type=PersistenceContextType.EXTENDED)
    EntityManager em;

    Query unassignedQuery;

    @PostConstruct
    public void init() {
        unassignedQuery =
            em.createQuery("SELECT e " +
                           "FROM Employee e " +
                           "WHERE e.projects IS EMPTY");
    }
    public List findEmployeesWithoutProjects() {
        return unassignedQuery.getResultList();
    }

    // ...
}
```

## Working with Query Results

The *result type* of a query is determined by the expressions listed in the SELECT clause of the query. If the result type of a query is the `Employee` entity, then executing `getResultList()` will result in a collection of zero or more `Employee` entity instances. There is a wide variety of results possible depending on the makeup of the query. The following are just some of the types that may result from JPQL queries:

- Basic types, such as `String`, the primitive types, and JDBC types
- Entity types
- An array of `Object`
- User-defined types created from a constructor expression

For developers used to JDBC, the most important thing to remember when using the `Query` interface is that the results are not encapsulated in a `ResultSet`. The collection or single result corresponds directly to the result type of the query.

Whenever an entity instance is returned, it becomes managed by the active persistence context. If that entity instance is modified and the persistence context is part of a transaction, then the changes will be persisted to the database. The only exception to this rule is the use of transaction-scoped entity managers outside of a transaction. Any query executed in this situation returns detached entity instances instead of managed entity instances. To make changes on these detached entities, they must first be merged into a persistence context before they can be synchronized with the database.

A consequence of the long-term management of entities with application-managed and extended persistence contexts is that executing large queries will cause the persistence context to grow as it stores all of the managed entity instances that are returned. If many of these persistence contexts are holding onto large numbers of managed entities for long periods of time, then memory use may become a concern. The `clear()` method of the `EntityManager` interface may be used to clear application-managed and extended persistence contexts, removing unnecessary managed entities.

### Optimizing Read-Only Queries

When the query results will not be modified, queries using transaction-scoped entity managers outside of a transaction are typically more efficient than queries executed within a transaction when the result type is an entity. When query results are prepared within a transaction, the persistence provider has to take steps to convert the results into managed entities. This usually entails taking a snapshot of the data for each entity in order to have a baseline to compare against when the transaction is committed. If the managed entities are never modified, then the effort of converting the results into managed entities is wasted.

Outside of a transaction, in some circumstances the persistence provider may be able to optimize the case where the results will be detached immediately. Therefore it can avoid the overhead of creating the managed versions. Note that this technique does not work on application-managed or extended entity managers, since their persistence context outlives the transaction. Any query result from this type of persistence context may be modified for later synchronization to the database even if there is no transaction.

When encapsulating query operations behind a stateless session façade, the easiest way to execute non-transactional queries is to use the `NOT_SUPPORTED` transaction attribute for the session bean method. This will cause any active transaction to be suspended, forcing the query results to be detached and enabling this optimization. Listing 6-16 shows an example of this technique.

**Listing 6-16.** *Executing a Query Outside of a Transaction*

```
@Stateless
public class QueryServiceBean implements QueryService {
    @PersistenceContext(unitName="EmployeeService")
    EntityManager em;

    @TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
    public List findAllDepartmentsDetached() {
        return em.createQuery("SELECT d FROM Department d")
                 .getResultList();
    }

    // ...
}
```

## Special Result Types

The array of `Object` result occurs whenever a query involves more than one expression in the SELECT clause. Common examples include projection of entity fields and aggregate queries where grouping expressions or multiple functions are used. Listing 6-17 revisits the menu generator from Listing 6-14 using a projection query instead of returning full `Employee` entity instances. Each element of the `List` is cast to an array of `Object` that is then used to extract the employee and department name information.

**Listing 6-17.** *Handling Multiple Result Types*

```
public void displayProjectEmployees(String projectName) {
    List result = em.createQuery("SELECT e.name, e.department.name " +
                                 "FROM Project p JOIN p.employees e " +
                                 "WHERE p.name = ?1 " +
                                 "ORDER BY e.name")
                    .setParameter(1, projectName)
                    .getResultList();
    int count = 0;
    for (Iterator i = result.iterator(); i.hasNext();) {
        Object[] values = (Object[]) i.next();
        System.out.println(++count + ": " + values[0] + ", " + values[1]);
    }
}
```

Constructor expressions provide developers with a way to map array of `Object` result types to custom objects. Typically this is used to convert the results into JavaBean-style classes that provide getters for the different returned values. This makes the results easier to work with and makes it possible to use the results directly in an environment such as JavaServer Faces without additional translation.

A constructor expression is defined in JPQL using the NEW operator in the SELECT clause. The argument to the NEW operator is the fully qualified name of the class that will be instantiated to hold the results for each row of data returned. The only requirement on this class is that it has a constructor with arguments matching the exact type and order that will be specified in the query. Listing 6-18 shows an `EmpMenu` class defined in the package `example` that could be used to hold the results of the query that was executed in Listing 6-17.

**Listing 6-18.** *Defining a Class for Use in a Constructor Expression*

```
package example;

public class EmpMenu {
    private String employeeName;
    private String departmentName;

    public EmpMenu(String employeeName, String departmentName) {
        this.employeeName = employeeName;
        this.departmentName = departmentName;
    }

    public String getEmployeeName() { return employeeName; }
    public String getDepartmentName() { return departmentName; }
}
```

Listing 6-19 shows the same example as Listing 6-17 using the fully qualified `EmpMenu` class name in a constructor expression. Instead of working with array indexes, each result is cast to the `EmpMenu` class and used like a regular Java object.

**Listing 6-19.** *Using Constructor Expressions*

```
public void displayProjectEmployees(String projectName) {
    List result =
        em.createQuery("SELECT NEW example.EmpMenu(e.name, e.department.name) " +
                       "FROM Project p JOIN p.employees e " +
                       "WHERE p.name = ?1 " +
                       "ORDER BY e.name")
           .setParameter(1, projectName)
           .getResultList();
    int count = 0;
    for (Iterator i = result.iterator(); i.hasNext();) {
        EmpMenu menu = (EmpMenu) i.next();
        System.out.println(++count + ": " + menu.getEmployeeName() + ", " +
                           menu.getDepartmentName());
    }
}
```

## Query Paging

Large result sets from queries are often a problem for many applications. In cases where it would be overwhelming to display the entire result set, or if the application medium makes displaying many rows inefficient (web applications, in particular), applications must be able to display ranges of a result set and provide users with the ability to control the range of data that they are viewing. The most common form of this technique is to present the user with a fixed-size table that acts as a sliding window over the result set. Each increment of results displayed is called a *page*, and the process of navigating through the results is called *pagination*.

Efficiently paging through result sets has long been a challenge for both application developers and database vendors. Before support existed at the database level, a common technique was to first retrieve all of the primary keys for the result set and then issue separate queries for the full results using ranges of primary key values. Later, database vendors added the concept of logical row number to query results, guaranteeing that so long as the result was ordered, the row number could be relied on to retrieve portions of the result set. More recently, the JDBC specification has taken this even further with the concept of scrollable result sets, which can be navigated forwards and backwards as required.

The `Query` interface provides support for pagination via the `setFirstResult()` and `setMaxResults()` methods. These methods specify the first result to be received (numbered from zero) and the maximum number of results to return relative to that point. A persistence provider may choose to implement support for this feature in a number of different ways, as not all databases benefit from the same approach. It's a good idea to become familiar with how your vendor approaches pagination and what level of support exists in the target database platform for your application.

---

■**Caution** The `setFirstResult()` and `setMaxResults()` methods should not be used with queries that join across collection relationships (one-to-many and many-to-many) because these queries may return duplicate values. The duplicate values in the result set make it impossible to use a logical result position.

---

To better illustrate pagination support, consider the stateful session bean shown in Listing 6-20. Once created, it is initialized with the name of a query to count the total results and the name of a query to generate the report. When results are requested, it uses the page size and current page number to calculate the correct parameters for the `setFirstResult()` and `setMaxResults()` methods. The total number of results possible is calculated by executing the count query. By using the `next()`, `previous()`, and `getCurrentResults()` methods, presentation code can page through the results as required. If this session bean were bound into an HTTP session, it could be directly used by a JSP or JavaServer Faces page presenting the results in a data table.

**Listing 6-20.** *Stateful Session Report Pager*

```
@Stateful
public class ResultPagerBean implements ResultPager {
    @PersistenceContext(unitName="QueryPaging")
    private EntityManager em;

    private String reportQueryName;
    private int currentPage;
    private int maxResults;
    private int pageSize;

    public int getPageSize() {
        return pageSize;
    }

    public int getMaxPages() {
        return maxResults / pageSize;
    }

    public void init(int pageSize, String countQueryName,
                     String reportQueryName) {
        this.pageSize = pageSize;
        this.reportQueryName = reportQueryName;
        maxResults = (Long) em.createNamedQuery(countQueryName)
                              .getSingleResult();
        maxResults = resultCount.longValue();
        currentPage = 0;
    }
    public List getCurrentResults() {
        return em.createNamedQuery(reportQueryName)
                 .setFirstResult(currentPage * pageSize)
                 .setMaxResults(pageSize)
                 .getResultList();
    }

    public void next() {
        currentPage++;
    }

    public void previous() {
        currentPage--;
        if (currentPage < 0) {
            currentPage = 0;
        }
    }

    public int getCurrentPage() {
        return currentPage;
    }

    public void setCurrentPage(int currentPage) {
        this.currentPage = currentPage;
    }

    @Remove
    public void finished() {
    }
}
```

## Queries and Uncommitted Changes

Executing queries against entities that have been created or changed in a transaction is a topic that requires special consideration. As we discussed in Chapter 5, the persistence provider will attempt to minimize the number of times the persistence context must be flushed within a transaction. Optimally this will occur only once, when the transaction commits. While the transaction is open and changes are being made, the provider relies on its own internal cache synchronization to ensure that the right version of each entity is used in entity manager operations. At most the provider may have to read new data from the database in order to fulfill a request. All entity operations other than queries can be satisfied without flushing the persistence context to the database.

Queries are a special case because they are executed directly as SQL against the database. Because the database executes the query and not the persistence provider, the active persistence context cannot usually be consulted by the query. As a result, if the persistence context has not been flushed and the database query would be impacted by the changes pending in the persistence context, incorrect data is likely to be retrieved from the query. The entity manager find() operation, on the other hand, always checks the persistence context before going to the database, so this is not a concern.

The good news is that by default, the persistence provider will ensure that queries are able to incorporate pending transactional changes in the query result. It might accomplish this by flushing the persistence context to the database, or it might leverage its own runtime information to ensure the results are correct.

And yet, there are times when having the persistence provider ensure query integrity is not necessarily the behavior we need. The problem is that it is not always easy for the provider to determine the best strategy to accommodate the integrity needs of a query. There is no way the provider can logically determine at a fine-grained level which objects have changed and therefore need to be incorporated into the query results. If the provider solution to ensuring query integrity is to flush the persistence context to the database, then you might have a performance problem if this is a frequent occurrence.

To put this issue in context, consider a message board application, which has modeled conversation topics as Conversation entities. Each Conversation entity refers to one or more messages represented by a Message entity. Periodically, conversations are archived when the last message added to the conversation is more than 30 days old. This is accomplished by changing the status of the Conversation entity from “ACTIVE” to “INACTIVE”. The two queries to obtain the list of active conversations and the last message date for a given conversation are shown in Listing 6-21.

**Listing 6-21.** *Conversation Queries*

```
@NamedQueries({
    @NamedQuery(name="findActiveConversations",
                query="SELECT c " +
                      "FROM Conversation c " +
                      "WHERE c.status = 'ACTIVE'"),
    @NamedQuery(name="findLastMessageDate",
                query="SELECT MAX(m.postingDate) " +
                      "FROM Conversation c JOIN c.messages m " +
                      "WHERE c = :conversation")
})
```

Listing 6-22 shows the session bean method used to perform this maintenance, accepting a Date argument that specifies the minimum age for messages in order to still be considered an active conversation. In this example, we see that two queries are being executed. The “findAllActiveConversations” query collects all of the active conversations, while the “findLastMessageDate” returns the last date that a message was added to a Conversation entity. As the code iterates over the Conversation entities, it invokes the “findLastMessage-Date” query for each one. As these two queries are related, it is reasonable for a persistence provider to assume that the results of the “findLastMessageDate” query will depend on the changes being made to the Conversation entities. If the provider ensures the integrity of the “findLastMessageDate” query by flushing the persistence context, this could become a very expensive operation if hundreds of active conversations are being checked.

**Listing 6-22.** *Archiving Conversation Entities*

```
@Stateless
public class ConversationMaintenanceBean implements ConversationMaintenance {
    @PersistenceContext(unitName="MessageBoard")
    EntityManager em;

    public void archiveConversations(Date minAge) {
        List<Conversation> active = (List<Conversation>)
            em.createNamedQuery("findActiveConversations")
              .getResultList();
        Query maxAge = em.createNamedQuery("findLastMessageDate");
        for (Conversation c : active) {
            maxAge.setParameter("conversation", c);
            Date lastMessageDate  = (Date) maxAge.getSingleResult();
            if (lastMessageDate.before(minAge)) {
                c.setStatus("INACTIVE");
            }
        }
    }

    // ...
}
```

To give developers more control over the integrity requirements of queries, the `EntityManager` and `Query` interfaces support a `setFlushMode()` method to set the *flush mode*, an indicator to the provider how it should handle pending changes and queries. There are two possible flush mode settings, `AUTO` and `COMMIT`, which are defined by the `FlushModeType` enumerated type. The default setting is `AUTO`, which means that the provider should ensure that pending transactional changes are included in query results. If a query might overlap with changed data in the persistence context, then this setting will ensure that the results are correct.

The `COMMIT` flush mode tells the provider that queries don't overlap with changed data in the persistence context, so it does not need to do anything in order to get correct results. Depending on how the provider implements its query integrity support, this might mean that it does not have to flush the persistence context before executing a query since you have indicated that there is nothing in memory that will be queried from the database.

Although the flush mode is set on the entity manager, the flush mode is really a property of the persistence context. For transaction-scoped entity managers, that means the flush mode has to be changed in every transaction. Extended and application-managed entity managers will preserve their flush-mode setting across transactions.

Setting the flush mode on the entity manager applies to all queries, while setting the flush mode for a query limits the setting to that scope. Setting the flush mode on the query overrides the entity manager setting as you would expect. If the entity manager setting is `AUTO` and one query has the `COMMIT` setting, then the provider will guarantee query integrity for all of the queries other than the one with the COMMIT setting. Likewise if the entity manager setting is COMMIT and one query has an AUTO setting, then only the query with the AUTO setting is guaranteed to incorporate pending changes from the persistence context.

Generally speaking, if you are going to execute queries in transactions where data is being changed, AUTO is the right answer. If you are concerned about the performance implications of ensuring query integrity, consider changing the flush mode to COMMIT on a per-query basis. Changing the value on the entity manager, while convenient, can lead to problems if more queries are added to the application later and they require AUTO semantics.

Coming back to the example at the start of this section, we can set the flush mode on the Query object for the “findLastMessageDate” query to COMMIT because it does not need to see the changes being made to the Conversation entities. The following fragment shows how this would be accomplished for the archiveConversations() method shown in Listing 6-22:

```
public void archiveConversations(Date minAge) {
    // ...
    Query maxAge = em.createNamedQuery("findLastMessageDate");
    maxAge.setFlushMode(FlushModeType.COMMIT);
    // ...
}
```

## Bulk Update and Delete

Like their SQL counterparts, JPQL bulk update and delete statements are designed to make changes to large numbers of entities in a single operation without requiring the individual entities to be retrieved and modified using the entity manager. Unlike SQL, which operates on tables, JPQL update and delete statements must take the full range of mappings for the entity into account. These operations are challenging for vendors to implement correctly, and as a result, there are restrictions on the use of these operations that must be well understood by developers.

The full syntax for UPDATE and DELETE statements is described in Chapter 7. The following sections will describe how to use these operations effectively and the issues that may result when used incorrectly.

### Using Bulk Update and Delete

Bulk update of entities is accomplished with the UPDATE statement. This statement operates on a single entity type and sets one or more single-valued properties of the entity (either a state field or a single-valued association) subject to the conditions in the WHERE clause. In terms of syntax, it is nearly identical to the SQL version with the exception of using entity expressions instead of tables and columns. Listing 6-23 demonstrates using a bulk update statement. Note that the use of the REQUIRES_NEW transaction attribute type is significant and will be discussed following the examples.

**Listing 6-23.** *Bulk Update of Entities*

```
@Stateless
public class EmployeeServiceBean implements EmployeeService {
    @PersistenceContext(unitName="BulkQueries")
    EntityManager em;

    @TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
    public void assignManager(Department dept, Employee manager) {
         em.createQuery("UPDATE Employee e " +
                        "SET e.manager = ?1 " +
                        "WHERE e.department = ?2 ")
           .setParameter(1, manager)
           .setParameter(2, dept)
           .executeUpdate();
    }
}
```

Bulk removal of entities is accomplished with the DELETE statement. Again, the syntax is the same as the SQL version except that the target in the FROM clause is an entity instead of a table and the WHERE clause is composed of entity expressions instead of column expressions. Listing 6-24 demonstrates bulk removal of entities.

**Listing 6-24.** *Bulk Removal of Entities*

```
@Stateless
public class ProjectServiceBean implements ProjectService {
    @PersistenceContext(unitName="BulkQueries")
    EntityManager em;

    @TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
    public void removeEmptyProjects() {
         em.createQuery("DELETE FROM Project p " +
                        "WHERE p.employees IS EMPTY ")
           .executeUpdate();
    }
}
```

The first issue for developers to consider when using these statements is that the persistence context is not updated to reflect the results of the operation. Bulk operations are issued as SQL against the database, bypassing the in-memory structures of the persistence context. Therefore updating the salary of all of the employees will not change the current values for any entities managed in memory as part of a persistence context. The developer can rely only on entities retrieved after the bulk operation completes.

For developers using transaction-scoped persistence contexts, this means that the bulk operation should either execute in a transaction all by itself or be the first operation in the transaction. Running the bulk operation in its own transaction is the preferred approach as it minimizes the chance of the developer accidentally fetching data before the bulk change occurs. Executing the bulk operation and then working with entities after it completes is also safe, because then any `find()` operation or query will go to the database to get current results. The examples in Listing 6-23 and Listing 6-24 used the `REQUIRES_NEW` transaction attribute to ensure that the bulk operations occurred within their own transactions.

A typical strategy for persistence providers dealing with bulk operations is to invalidate any in-memory cache of data related to the target entity. This forces data to be fetched from the database the next time it is required. How much cached data gets invalidated depends on the sophistication of the persistence provider. If the provider can detect that the update impacts only a small range of entities, then those specific entities may be invalidated, leaving other cached data in place. Such optimizations are limited, however, and if the provider cannot be sure of the scope of the change, then the entire cache must be invalidated. This can have performance impacts on the application if bulk changes are a frequent occurrence.

---

■ **Caution** SQL update and delete operations should never be executed on tables mapped by an entity. The JPQL operations tell the provider what cached entity state must be invalidated in order to remain consistent with the database. Native SQL operations bypass such checks and can quickly lead to situations where the in-memory cache is out of date with respect to the database.

---

The danger present in bulk operations and the reason they must occur first in a transaction is that any entity actively managed by a persistence context will remain that way, oblivious to the actual changes occurring at the database level. The active persistence context is separate and distinct from any data cache that the provider may use for optimizations. Consider the following sequence of operations:

1. A new transaction starts.
2. Entity A is created by calling `persist()` to make the entity managed.
3. Entity B is retrieved from a `find()` operation and modified.
4. A bulk remove deletes entity A.
5. A bulk update changes the same properties on entity B that were modified in step 3.
6. The transaction commits.

What should happen to entities A and B in this sequence? In the case of entity A, the provider has to assume that the persistence context is correct and so will still attempt to insert the new entity even though it should have been removed. In the case of entity B, again the provider has to assume that managed version is the correct version and will attempt to update the version in the database, undoing the bulk update change.

This brings us to the issue of extended persistence contexts. Bulk operations and extended persistence contexts are a particularly dangerous combination because the persistence context survives across transaction boundaries, but the provider will never refresh the persistence context to reflect the changed state of the database after a bulk operation has completed. When the extended persistence context is next associated with a transaction, it will attempt to synchronize its current state with the database. Since the managed entities in the persistence context are now out of date with respect to the database, any changes made since the bulk operation could result in incorrect results being stored. In this situation, the only option is to refresh the entity state or ensure that the data is versioned in such a way that the incorrect change can be detected. Locking strategies and refreshing of entity state are discussed in Chapter 9.

## Bulk Delete and Relationships

In our discussion of the `remove()` operation in the previous chapter, we emphasized that relationship maintenance is always the responsibility of the developer. The only time a cascading remove occurs is when the `REMOVE` cascade option is set for a relationship. Even then, the persistence provider won't automatically update the state of any managed entities that refer to the removed entity. As we are about to see, the same requirement holds true when using DELETE statements as well.

A DELETE statement in JPQL corresponds more or less to a DELETE statement in SQL. Writing the statement in JPQL gives you the benefit of working with entities instead of tables, but the semantics are exactly the same. This has implications in how applications must write DELETE statements in order to ensure that they execute correctly and leave the database in a consistent state.

DELETE statements do not cascade to related entities. Even if the `REMOVE` cascade option is set on a relationship, it will not be followed. It is your responsibility to ensure that relationships are correctly updated with respect to the entities that have been removed. The persistence provider also has no control over constraints in the database. If you attempt to remove data that is the target of a foreign key relationship in another table, you will get a referential integrity constraint violation in return.

Let's look at an example that puts these issues in context. Consider, for example, that a company wishes to reorganize its department structure. We want to delete a number of departments and then assign the employees to new departments. The first step is to delete the old departments, so the following statement is to be executed:

```
DELETE FROM Department d
WHERE d.name IN ('CA13', 'CA19', 'NY30')
```

This is a straightforward operation. We want to remove the department entities that match the given list of names using a DELETE statement instead of querying for the entities and using the `remove()` operation to dispose of them. But when this query is executed, a `PersistenceException` exception is thrown, reporting that a foreign key integrity constraint has been violated. Therefore, another table has a foreign key reference to one of the rows we are trying to delete. Checking the database, we see that the table mapped by the `Employee` entity has a foreign key constraint against the table mapped by the `Department` entity. Since the foreign key value in the `Employee` table is not `NULL`, the parent key from the `Department` table can't be removed.

Therefore we need to first update the `Employee` entities in question to make sure that they do not point to the department we are trying to delete:

```
UPDATE Employee e
SET e.department = null
WHERE e.department.name IN ('CA13', 'CA19', 'NY30')
```

With this change the original DELETE statement will work as expected. Now consider what would have happened if the integrity constraint had not been in the database. The DELETE operation would have completed successfully, but the foreign key values would still be sitting in the `Employee` table. The next time the persistence provider tried to load the `Employee` entities with dangling foreign keys, it would be unable to resolve the target entity. The outcome of this operation is vendor-specific but will most likely lead to a `PersistenceException` exception being thrown, complaining of the invalid relationship.

## Query Hints

Query hints are the Java Persistence API extension point for vendor-specific query features. A hint is simply a string name and object value. The meaning of both the name and value is entirely up to the persistence provider. Every query may be associated with any number of hints, set either in persistence unit metadata as part of the `@NamedQuery` annotation, or on the `Query` interface itself using the `setHint()` method.

We left query hints until the end of this chapter because they are the only feature in the query API that has no standard usage. Everything about hints is vendor-specific. The only guarantee provided by the specification is that providers must ignore hints that they do not understand. Listing 6-25 demonstrates the "toplink.cache-usage" hint supported by the Reference Implementation of the Java Persistence API to indicate that the cache should not be checked when reading an `Employee` from the database. Unlike the `refresh()` method of the `EntityManager` interface, this hint will not cause the query result to override the current cached value.

**Listing 6-25.** *Using Query Hints*

```
public Employee findEmployeeNoCache(int empId) {
    Query q = em.createQuery("SELECT e FROM Employee e WHERE e.id = ?1");
    // force read from database
    q.setHint("toplink.cache-usage", "DoNotCheckCache");
    q.setParameter(1, empId);
    try {
        return (Employee) q.getSingleResult();
    } catch (NoResultException e) {
        return null;
    }
}
```

If this query were to be executed frequently, a named query would be more efficient. The following named query definition incorporates the cache hint used earlier:

```
@NamedQuery(name="findEmployeeNoCache",
            query="SELECT e FROM Employee e WHERE e.id = :empId",
            hints={@QueryHint(name="toplink.cache-usage", value="DoNotCheckCache")})
```

The `hints` element accepts an array of `@QueryHint` annotations, allowing any number of hints to be set for a query.

## Query Best Practices

The typical application using the Java Persistence API is going to have many queries defined. It is the nature of enterprise applications that information is constantly being queried from the database, for everything from complex reports to drop-down lists in the user interface. Therefore efficiently using queries can have a major impact on your application as a whole.

### Named Queries

First and foremost, we recommend named queries whenever possible. Persistence providers will often take steps to precompile JPQL named queries to SQL as part of the deployment or initialization phase of an application. This avoids the overhead of continuously parsing JPQL and generating SQL. Even with a cache for converted queries, dynamic query definition will always be less efficient than using named queries.

Named queries also enforce the best practice of using query parameters. Query parameters help to keep the number of distinct SQL strings parsed by the database to a minimum. Since databases typically keep a cache of SQL statements on hand for frequently accessed queries, this is an essential part of ensuring peak database performance.

As we discussed in the Dynamic Query Definition section, query parameters also help to avoid security issues caused by concatenating values into query strings. For applications exposed to the web, security has to be a concern at every level of an application. You can either spend a lot of effort trying to validate input parameters, or you can use query parameters and let the database do the work for you.

When naming queries, decide on a naming strategy early in the application development cycle with the understanding that the query namespace is global for each persistence unit. Collisions between query names are likely to be a common frustration if there is no established naming pattern.

Finally, using named queries allows for JPQL queries to be overridden with SQL queries or even with vendor-specific languages and expression frameworks. For applications migrating from an existing object-relational mapping solution, it is quite likely that the vendor will provide some support for invoking their existing query solution using the named query facility in the Java Persistence API. We will discuss SQL named queries in Chapter 9.

### Report Queries

If you are executing queries that return entities for reporting purposes and have no intention of modifying the results, consider executing queries using a transaction-scoped entity manager but outside of a transaction. The persistence provider may be able to detect the lack of a transaction and optimize the results for detachment, often by skipping some of the steps required to create an interim managed version of the entity results.

Likewise, if an entity is expensive to construct due to eager relationships or a complex table mapping, consider selecting individual entity properties using a projection query instead of retrieving the full entity result. If all you need is the name and office phone number for 500 employees, selecting only those two fields is likely to be far more efficient than fully constructing 1,000 entity instances.

## Query Hints

It is quite likely that vendors will entice you with a variety of hints to enable different performance optimizations for queries. Query hints may well be an essential tool in meeting your performance expectations. We strongly advise, however, that you resist the urge to embed query hints in your application code. The ideal location for query hints is in an XML mapping file (which we will be describing in Chapter 10), or at the very least as part of a named query definition. Hints are often highly dependent on the target platform and may well change over time as different aspects of the application impact the overall balance of performance. Keep hints decoupled from your code if at all possible.

## Stateless Session Beans

We tried to demonstrate as many examples as possible in the context of a stateless session bean method, as we believe that this is the best way to organize queries in a Java EE application. Using the stateless session bean has a number of benefits over simply embedding queries all over the place in application code:

- Clients can execute queries by invoking an appropriately named business method instead of relying on a cryptic query name or multiple copies of the same query string.
- Stateless session bean methods can optimize their transaction usage depending on whether or not the results need to be managed or detached.
- Using a transaction-scoped persistence context ensures that large numbers of entity instances don't remain managed long after they are needed.
- For existing entity bean applications, the stateless session bean is the ideal vehicle for migrating finder queries away from the entity bean home interface. We will discuss this technique in Chapter 13.

This is not to say that other components are unsuitable locations for queries, but stateless session beans are a well-established best practice for hosting queries in the Java EE environment.

## Bulk Update and Delete

If bulk update and delete operations must be used, ensure that they are executed only in an isolated transaction where no other changes are being made. There are many ways in which these queries can negatively impact an active persistence context. Interweaving these queries with other non-bulk operations requires careful management by the application.

Entity versioning and locking requires special consideration when bulk update operations are used. Bulk delete operations can have wide ranging ramifications depending on how well the persistence provider can react and adjust entity caching in response. Therefore we view bulk update and delete operations as being highly specialized, to be used with care.

## Provider Differences

Take time to become familiar with the SQL that your persistence provider generates for different JPQL queries. Although understanding SQL is not necessary for writing JPQL queries, knowing what happens in response to the various JPQL operations is an essential part of performance tuning. Joins in JPQL are not always explicit, and you may find yourself surprised at the complex SQL generated for a seemingly simple JPQL query.

The benefits of features such as query paging are also dependent on the approach used by your persistence provider. There are a number of different techniques that can be used to accomplish pagination, many of which suffer from performance and scalability issues. Because the Java Persistence API can't dictate a particular approach that will work well in all cases, become familiar with the approach used by your provider and whether or not it is configurable.

Finally, understanding the provider strategy for when and how often it flushes the persistence context is necessary before looking at optimizations such as changing the flush mode. Depending on the caching architecture and query optimizations used by a provider, changing the flush mode may or may not make a difference to your application.

## Summary

We began this chapter with an introduction to JPQL, the query language defined by the Java Persistence API. We briefly discussed the origins of JPQL and its role in writing queries that interact with entities. We also provided an overview of major JPQL features for developers already experienced with SQL or previous versions of EJB QL.

In the discussion on executing queries, we introduced the methods for defining queries both dynamically at runtime and statically as part of persistence unit metadata. We looked at the `Query` interface and the types of query results possible using JPQL. We also looked at parameter binding, strategies for handling large result sets and how to ensure that queries in transactions with modified data complete successfully.

In the section on bulk update and delete we looked at how to execute these types of queries and how to ensure that they are used safely by the application. We provided details on how persistence providers deal with bulk operations and the impact that they have on the active persistence context.

We ended our discussion of query features with a look at query hints. We showed how to specify hints and provided an example using hints supported by the Reference Implementation of the Java Persistence API.

Finally, we summarized our view of best practices relating to queries, looking at named queries, different strategies for the various query types, as well as the implementation details that need to be understood for different persistence providers.

In the next chapter, we will continue to focus on queries by examining JPQL in detail.