JPA Queries

Revision: v2014-05-06

Built on: 2014-06-29 08:57 EST

This presentation provides information covering JPA Queries, JPA Query Language, and the JPA Criteria API.

Purpose

1. Goals

- Provide breadth coverage of JPA Queries to demonstrate options available for accessing information from a relational database using an EntityManager.

2. Objectives

At the completion of this topic, the student shall

- have an understanding of:
  - Query Construction
    - Value, ResultClass, and Entity Queries
    - Typed Queries
    - Dynamic and Named Queries
    - Single and Multiple Results
    - Parameters
    - Paging
    - Locking
  - Supported Query Languages
    - JPA Query Language
    - Native SQL
    - Java-based Criteria API
- be able to:
  - Form a query using...
    - JPA-QL
    - Native SQL
    - Criteria API
  - Use a query to locate specific properties for one or more entities that match a criteria
  - Use a query to locate specific entities that match a criteria
  - Use paging within queries to handle large data sets
  - Use pessimistic locking to better support database consistency

Part I. General Queries

JPA Query Types

Three fundamental query types within JPA

- JPA Query Language (JPA) - entity/property/relationship-based
- Native SQL - table/column-based
- Criteria API - entity/property/relationship-based using Java classes

1.1. JPA Query Language (JPA-QL) Queries

- Access to the *entity* model using a SQL-like text query language
- Queries expressed using entities, properties, and relationships
- Pros
  - More concise (than other query forms)
  - Familiar to SQL users
  - Abstracts query away from table, column, primary key, and relationship mapping
  - Can be defined within XML deployment descriptors
  - Produces portable SQL
- Cons
  - Not (overly) type-safe
  - No help from Java compiler in constructing query expression
  - Don't find out most errors until runtime

Figure 1.1. Building a JPA Query using JPA-QL

```
String jpaqlString =
    "select c from Customer c " +
    "where c.firstName = :firstName " +
    "order by c.lastName ASC";
//use query string to build typed JPA-QL query
TypedQuery<Customer> query = em
        .createQuery(jpaqlString,Customer.class);
```

- "c" is part of root query
- "c" represents rows from Customer entity table(s)
- "c.lastName" is path off root term
- ":firstName" is parameter placeholder
- "c.firstName" is path off root term
- "Customer.class" type parameter allows for a type-safe return result

Figure 1.2. Executing a JPA Query (built from JPA-QL)

```
//at this point we are query-type agnostic
List<Customer> customers = query
        .setParameter("firstName", "thing")
        .getResultList();
log.info("result=" + customers);
assertEquals("unexpected number of results", 2, customers.size());
```

```

select
    customer0_.CUSTOMER_ID as CUSTOMER1_3_,
    customer0_.FIRST_NAME as FIRST2_3_,
    customer0_.LAST_NAME as LAST3_3_
from JPAQL_CUSTOMER customer0_
where customer0_.FIRST_NAME=?
order by customer0_.LAST_NAME ASC

-result=[firstName=thing, lastName=one, firstName=thing, lastName=two]
```

- Placeholder is replaced by runtime parameter
- Zero-or-more results are requested
- Entities returned are managed

Figure 1.3. Condensing the JPA-QL Query

```
List<Customer> customers = em.createQuery(
    "select c from Customer c " +
    "where c.firstName = :firstName " +
    "order by c.lastName ASC",
    Customer.class)
        .setParameter("firstName", "thing")
        .getResultList();
log.info("result=" + customers);
assertEquals("unexpected number of results", 2, customers.size());
```

1.2. Native SQL Queries

- Access to power of working with native SQL
- Pros
  - Provides full access to native SQL power
  - Provides full access to database-vendor SQL extensions
  - Easy to see when native SQL is being used within application -- target for portability review
  - Ability to produce managed entity as result of query
- Cons
  - Portability of SQL not addressed by JPA
  - Not type-safe
  - No help from Java compiler in constructing query expression
  - Don't find out most errors until runtime

Figure 1.4. Building a JPA Query using Native SQL

```
Table table = Customer.class.getAnnotation(Table.class);
```

```
String sqlString =
  "select c.CUSTOMER_ID, c.FIRST_NAME, c.LAST_NAME " +
  String.format("from %s c ", table.name()) +
  "where c.FIRST_NAME = ? " +
  "order by c.LAST_NAME ASC";
//use query string to build query
Query query = em.createNativeQuery(sqlString,Customer.class);
```

- "c" represents rows in table
- specific columns (or *) are return for each row
- "?" marks a positional parameter -- non-portable to use named parameters in native SQL queries
- TypedQuery<T>s not supported in native SQL queries because of a conflict with legacy JPA 1.0 API

Figure 1.5. Executing a JPA Query (built from Native SQL)

```
//at this point we are query-type agnostic (mostly)
@SuppressWarnings("unchecked")
List<Customer> customers = query
        .setParameter(1, "thing")
        .getResultList();
log.info("result=" + customers);
assertEquals("unexpected number of results", 2, customers.size());
```

```

select
  c.CUSTOMER_ID,
  c.FIRST_NAME,
  c.LAST_NAME
from JPAQL_CUSTOMER c
where c.FIRST_NAME = ?
order by c.LAST_NAME ASC

-result=[firstName=thing, lastName=one, firstName=thing, lastName=two]
```

- Query execution similar to other query types
- User-provided SQL executed

Note

Legacy JPA 1.0 Native SQL query syntax already used the signature of passing in a Class for createNativeQuery(). In this context, it was an entity class that contained JPA mappings for the query -- not the returned entity type. This prevented createNativeQuery() from being updated to return a typed result in JPA 2.0.

Figure 1.6. Condensing the SQL Query

```
@SuppressWarnings("unchecked")
List<Customer> customers = em.createNativeQuery(
  "select c.CUSTOMER_ID, c.FIRST_NAME, c.LAST_NAME " +
  "from JPAQL_CUSTOMER c " +
  "where c.FIRST_NAME = ? " +
  "order by c.LAST_NAME ASC",
  Customer.class)
    .setParameter(1, "thing")
    .getResultList();
log.info("result=" + customers);
assertEquals("unexpected number of results", 2, customers.size());
```

1.2.1. SqlResultSetMappings

- Allow query to return mixture of managed entities and values
- DAOs can use value results to plugin transient aggregate properties in parent entity without pulling entire child entities back from database
    - e.g., total sales for clerk

Figure 1.7. NativeQuery with SqlResultSetMapping

```
@SuppressWarnings("unchecked")
List<Object[]> results = em.createNativeQuery(
    "select clerk.CLERK_ID, "
    + "clerk.FIRST_NAME, "
    + "clerk.LAST_NAME, "
    + "clerk.HIRE_DATE, "
    + "clerk.TERM_DATE, "
    + "sum(sales.amount) total_sales "
    + "from JPAQL_CLERK clerk "
    + "left outer join JPAQL_SALE_CLERK_LINK slink on clerk.CLERK_ID=slink.CLERK_ID "
    + "left outer join JPAQL_SALE sales on sales.SALE_ID=slink.SALE_ID "
    + "group by clerk.CLERK_ID, "
    + "clerk.FIRST_NAME, "
    + "clerk.LAST_NAME, "
    + "clerk.HIRE_DATE, "
    + "clerk.TERM_DATE "
    + "order by total_sales DESC",
    "Clerk.clerkSalesResult")
    .getResultList();
```

```
@Entity @Table(name="JPAQL_CLERK")
@SqlResultSetMappings({
  @SqlResultSetMapping(name = "Clerk.clerkSalesResult",
      entities={ @EntityResult(entityClass = Clerk.class )},
      columns={@ColumnResult(name = "total_sales")}
  )
})
public class Clerk {
```

Figure 1.8. Example NativeQuery with SqlResultSetMapping Output

```
for (Object[] result: results) {
  Clerk clerk = (Clerk) result[0];
  BigDecimal totalSales = (BigDecimal) result[1];
```

```
    log.info(String.format("%s, $ %s", clerk.getFirstName(), totalSales));
}
```

```
-Manny, $ 250.00
-Moe, $ 150.00
-Jack, $ null
```

Figure 1.9. NamedNativeQuery with SqlResultSetMapping

```
@Entity @Table(name="JPAQL_CLERK")
@NamedNativeQueries({
    @NamedNativeQuery(name = "Clerk.clerkSales", query =
            "select clerk.CLERK_ID, "
            + "clerk.FIRST_NAME, "
            + "clerk.LAST_NAME, "
            + "clerk.HIRE_DATE, "
            + "clerk.TERM_DATE, "
            + "sum(sales.amount) total_sales "
            + "from JPAQL_CLERK clerk "
            + "left outer join JPAQL_SALE_CLERK_LINK slink on clerk.CLERK_ID=slink.CLERK_ID "
            + "left outer join JPAQL_SALE sales on sales.SALE_ID=slink.SALE_ID "
            + "group by clerk.CLERK_ID, "
            + "clerk.FIRST_NAME, "
            + "clerk.LAST_NAME, "
            + "clerk.HIRE_DATE, "
            + "clerk.TERM_DATE "
            + "order by total_sales DESC",
            resultSetMapping="Clerk.clerkSalesResult")
})
@SqlResultSetMappings({
    @SqlResultSetMapping(name = "Clerk.clerkSalesResult",
            entities={ @EntityResult(entityClass = Clerk.class )},
            columns={@ColumnResult(name = "total_sales")}
    )
})
public class Clerk {
```

Figure 1.10. Example NamedNativeQuery with SqlResultSetMapping Usage

```
List<Object[]> results = em.createNamedQuery("Clerk.clerkSales").getResultList();
```

1.3. Criteria API Queries

- Somewhat parallel capability to JPAQL
- Build overall query using Java types (demonstrated here with "string accessors")
- Pros
  - Structure of query is type-safe
  - Allows object-level manipulation of the query versus manipulation of a query string
    - Useful when building total query based on runtime properties
- Cons
  - Complex -- looses familiarity with SQL

- Cannot be expressed in XML deployment descriptor
- Access to properties not type-safe (addressed by *Canonical Metamodel)*

Figure 1.11. Building a JPA Query using Criteria API

```
select c from Customer c
where c.firstName = :firstName
order by c.lastName ASC
```

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Customer> qdef = cb.createQuery(Customer.class);
Root<Customer> c = qdef.from(Customer.class);
qdef.select(c)
   .where(cb.equal(c.get("firstName"),
               cb.parameter(String.class,"firstName")))
   .orderBy(cb.asc(c.get("lastName")));
//build query from criteria definition
TypedQuery<Customer> query = em.createQuery(qdef);
```

- "CriteriaBuilder" used as starting point to build objects within the query tree
- "CriteriaQuery<T>" used to hold the definition of query
- "Root<T>" used to reference root level query terms
- "CriteriaBuilder.from()" used to designate the entity that represents root query term
  - Result used to create path references for query body
- "CriteriaBuilder.select()" officially lists the objects returned from query
- "CriteriaBuilder.where()" builds a decision predicate of which entities to include
- "CriteriaBuilder.equal()" builds an equals predicate for the where clause
- "Root<T>.get()" returns the property referenced in path expression
- "CriteriaBuilder.parameter()" builds a parameter placeholder within query. Useful with @Temporal date comparisons

Figure 1.12. Executing a JPA Query using Criteria API

```
//at this point we are query-type agnostic
List<Customer> customers = query
       .setParameter("firstName", "thing")
       .getResultList();
log.info("result=" + customers);
assertEquals("unexpected number of results", 2, customers.size());
```

```
   select
      customer0_.CUSTOMER_ID as CUSTOMER1_3_,
      customer0_.FIRST_NAME as FIRST2_3_,
      customer0_.LAST_NAME as LAST3_3_
   from JPAQL_CUSTOMER customer0_
   where customer0_.FIRST_NAME=?
   order by customer0_.LAST_NAME asc

 -result=[firstName=thing, lastName=one, firstName=thing, lastName=two]]
```

- Query execution identical to JPA-QL case

Figure 1.13. Condensing the Criteria API Query

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Customer> qdef = cb.createQuery(Customer.class);
Root<Customer> c = qdef.from(Customer.class);

List<Customer> customers = em.createQuery(qdef.select(c)
    .where(cb.equal(c.get("firstName"), "thing"))
    .orderBy(cb.asc(c.get("lastName"))))
    .getResultList();

log.info("result=" + customers);
assertEquals("unexpected number of results", 2, customers.size());
```

1.4. Strongly Typed Queries

- Previous Criteria API examples were string label based -- not type safe
- Criteria API provides means for stronger typing
- Strong typing permits automatic detection of model and query differences

1.4.1. Metamodel API

- Provides access to the persistent model backing each entity and its properties

Figure 1.14. Accessing JPA Metamodel

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Customer> qdef = cb.createQuery(Customer.class);
Root<Customer> c = qdef.from(Customer.class);
EntityType<Customer> c_ = c.getModel();

log.info(String.format("%7s, %10s:%-30s",
        c_.getPersistenceType(),
        c_.getName(),
        c_.getJavaType()));
for (Attribute<? super Customer, ?> p: c_.getAttributes()) {
    log.info(String.format("%7s, %10s:%-30s",
            p.getPersistentAttributeType(),
            p.getName(),
            p.getJavaType()));
}
```

```
-  ENTITY,   Customer:class ejava.jpa.examples.query.Customer
-   BASIC,  firstName:class java.lang.String
-   BASIC,         id:long
-   BASIC,   lastName:class java.lang.String
```

- JPA Metamodel provides access to
  - Entity structure
  - Entity database mapping

1.4.2. Query using JPA Metamodel

- Pros
  - Access properties in (a more) type-safe manner
- Cons
  - Complex
  - No compiler warning of entity type re-factoring

Figure 1.15. Building Query with JPA Metamodel

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Customer> qdef = cb.createQuery(Customer.class);
Root<Customer> c = qdef.from(Customer.class);
EntityType<Customer> c_ = c.getModel();
qdef.select(c) //we are returning a single root object
    .where(cb.equal(
            c.get(c_.getSingularAttribute("firstName", String.class)),
            cb.parameter(String.class,"firstName")))
    .orderBy(cb.asc(c.get(c_.getSingularAttribute("lastName", String.class))));
TypedQuery<Customer> query = em.createQuery(qdef);
```

- Access to properties within entities done through type-safe accessors

Figure 1.16. Executing Query with JPA Metamodel

```
//at this point we are query-type agnostic
List<Customer> customers = query
        .setParameter("firstName", "thing")
        .getResultList();
log.info("result=" + customers);
assertEquals("unexpected number of results", 2, customers.size());
```

```
select
    customer0_.CUSTOMER_ID as CUSTOMER1_3_,
    customer0_.FIRST_NAME as FIRST2_3_,
    customer0_.LAST_NAME as LAST3_3_
from JPAQL_CUSTOMER customer0_
where customer0_.FIRST_NAME=?
order by customer0_.LAST_NAME asc
-result=[firstName=thing, lastName=one, firstName=thing, lastName=two]
```

- Results identical to previous approaches

Figure 1.17. Condensing the JPA Metamodel-based Query

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Customer> qdef = cb.createQuery(Customer.class);
Root<Customer> c = qdef.from(Customer.class);
EntityType<Customer> c_ = c.getModel();
List<Customer> customers = em.createQuery(qdef.select(c)
    .where(cb.equal(
            c.get(c_.getSingularAttribute("firstName", String.class)), "thing"))
    .orderBy(cb.asc(c.get(c_.getSingularAttribute("lastName", String.class)))))
    .getResultList();

log.info("result=" + customers);
assertEquals("unexpected number of results", 2, customers.size());
```

1.4.3. Canonical Metamodel

- Complexities of metamodel cab be simplified using metamodel classes
- Pros
  - Easy, type-safe access to entity model
  - Java compiler can alert of mismatch between query and entity model
- Cons
  - Requires either manual construct or auto-generation of separate metamodel class

Figure 1.18. Example Canonical Metamodel

```
package ejava.jpa.examples.query;
import javax.persistence.metamodel.SingularAttribute;
import javax.persistence.metamodel.StaticMetamodel;

@StaticMetamodel(Customer.class)
public abstract class Customer_ {
    public static volatile SingularAttribute<Customer, Long> id;
    public static volatile SingularAttribute<Customer, String> lastName;
    public static volatile SingularAttribute<Customer, String> firstName;
}
```

- Construct or generate a canonical metamodel class to provide type-safe, easy access to properties

Figure 1.19. Building Query with Canonical Metamodel

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Customer> qdef = cb.createQuery(Customer.class);
Root<Customer> c = qdef.from(Customer.class);

qdef.select(c) //we are returning a single root object
    .where(cb.equal(
            c.get(Customer_.firstName),
            cb.parameter(String.class,"firstName")))
    .orderBy(cb.asc(c.get(Customer_.lastName)));
TypedQuery<Customer> query = em.createQuery(qdef);
```

- Use canonical metamodel class to provide type-safe, easy access to properties ("Customer_.firstName")

Figure 1.20. Executing Query with Canonical Metamodel

```
//at this point we are query-type agnostic
```

```
List<Customer> customers = query
        .setParameter("firstName", "thing")
        .getResultList();
log.info("result=" + customers);
assertEquals("unexpected number of results", 2, customers.size());
```

```
select
    customer0_.CUSTOMER_ID as CUSTOMER1_3_,
    customer0_.FIRST_NAME as FIRST2_3_,
    customer0_.LAST_NAME as LAST3_3_
from JPAQL_CUSTOMER customer0_
where customer0_.FIRST_NAME=?
order by customer0_.LAST_NAME asc
-result=[firstName=thing, lastName=one, firstName=thing, lastName=two]
```

- Result is identical to previous approaches

Figure 1.21. Condensing the Canonical Metamodel-based Query

```

CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Customer> qdef = cb.createQuery(Customer.class);
Root<Customer> c = qdef.from(Customer.class);

List<Customer> customers = em.createQuery(qdef.select(c)
    .where(cb.equal(c.get(Customer_.firstName),"thing"))
    .orderBy(cb.asc(c.get(Customer_.lastName))))
    .getResultList();
log.info("result=" + customers);
assertEquals("unexpected number of results", 2, customers.size());
```

- More work to get here but clean, result
- Type-safe - queries will not compile if entity changes

1.4.4. Generating Canonical Metamodel Classes

- Canonical Metamodel classes can be manually authoried or generated

Figure 1.22. Maven Dependency Can Generate Canonical Metamodel Classes

```
<!-- generates JPA metadata classes -->
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-jpamodelgen</artifactId>
    <version>1.1.1.Final</version>
    <scope>provided</scope>
</dependency>
```

Figure 1.23. Generated Source placed in target/generated-sources/annotations

```
`-- target
    |-- generated-sources
        `-- annotations
            `-- ejava
                `-- jpa
                    `-- examples
                        `-- query
                            |-- Clerk_.java
                            |-- Customer_.java
                            |-- Sale_.java
                            `-- Store_.java
```

Figure 1.24. Maven Plugin adds Generated Source to IDE Build Path

```
<!-- add generated JPA metamodel classes to classpath -->
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>build-helper-maven-plugin</artifactId>
    <version>1.8</version>
    <executions>
       <execution>
           <id>add-metamodel-classes</id>
           <phase>process-sources</phase>
           <goals>
               <goal>add-source</goal>
           </goals>
           <configuration>
               <sources>
                   <source>target/generated-sources/annotations</source>
               </sources>
           </configuration>
       </execution>
    </executions>
</plugin>
```

1.5. Summary

- Three basic forms for query expression
  - SqlResultSetMapping
  - JPAQL
  - Native SQL
  - Criteria API
    - String-based Accessors
    - Metamodel Accessors
    - Canonical Metamodel Accessors
- Entity model provides portability
  - JPAQL

  - Criteria API
- Native SQL provides direct access to
  - full power of SQL
  - full access to database-specific extensions
- Criteria API provides type-safe construct of query structure
- JPA Metamodel provides type-safe access to entity properties
- JPA Canonical Metamodel provides type-safe access to model-specific entity properties
  - Produces compilation error when query our of sync with entity model
  - Provides convenient access to model-specific properties

JPA Query Overview

2.1. EntityManager Query Methods

- Create query using JPA-QL String

```
javax.persistence.Query createQuery(String jpaql);
<T extends Object> javax.persistence.TypedQuery<T> createQuery(String jpaql, Class<T>);
```

- Create query using native SQL

```
javax.persistence.Query createNativeQuery(String sql);
javax.persistence.Query createNativeQuery(String sql, Class sqlMapping);
javax.persistence.Query createNativeQuery(String sql, String sqlMapping);
```

- Create query using Criteria API

```
javax.persistence.criteria.CriteriaBuilder getCriteriaBuilder();
javax.persistence.metamodel.Metamodel getMetamodel();
<T extends Object> javax.persistence.TypedQuery<T> createQuery(javax.persistence.criteria.CriteriaQuery<T>);
```

- Create query from Named Query

```
javax.persistence.Query createNamedQuery(String queryName);
<T extends java/lang/Object> javax.persistence.TypedQuery<T> createNamedQuery(String queryName, Class<T>);
```

2.2. Query.getSingleResult()

- Obtains exactly one result
- TypedQuery returns type-safe result

Figure 2.1. Get a Unique Object based on Query

```
TypedQuery<Store> query = em.createQuery(
        "select s from Store s where s.name='Big Al''s'", Store.class);
Store store = query.getSingleResult();
```

```
select
    store0_.STORE_ID as STORE1_4_,
    store0_.name as name2_4_
from ORMQL_STORE store0_
where store0_.name='Big Al''s'
```

Figure 2.2. Throws NoResultException when not Found

```
try {
    store = em.createQuery(
       "select s from Store s where s.name='A1 Sales'", Store.class)
       .getSingleResult();
}
catch (NoResultException ex) { ... }
```

Figure 2.3. Throws NonUniqueResultException when multiple Found

```
try {
    Clerk clerk = em.createQuery(
        "select c from Clerk c where lastName='Pep'", Clerk.class)
        .getSingleResult();
}
catch (NonUniqueResultException ex) { ... }
```

2.3. Query.getResultList

- Returns zero or more results
- TypedQuery returns type-safe result

Figure 2.4. Returns List of Results Based on Query

```
TypedQuery<Clerk> query = em.createQuery(
    "select c from Clerk c where lastName='Pep'", Clerk.class);
List<Clerk> clerks = query.getResultList();
assertTrue("unexpected number of clerks:" + clerks.size(), clerks.size() > 1);
for(Clerk c : clerks) {
    log.info("found clerk:" + c);
}
```

```
select
    clerk0_.CLERK_ID as CLERK1_0_,
    clerk0_.FIRST_NAME as FIRST2_0_,
    clerk0_.HIRE_DATE as HIRE3_0_,
    clerk0_.LAST_NAME as LAST4_0_,
    clerk0_.TERM_DATE as TERM5_0_
from JPAQL_CLERK clerk0_
where clerk0_.LAST_NAME='Pep'
...
-found clerk:firstName=Manny, lastName=Pep, hireDate=1970-01-01, termDate=null, sales(2)={1, 2, }
-found clerk:firstName=Moe, lastName=Pep, hireDate=1970-01-01, termDate=null, sales(1)={2, }
-found clerk:firstName=Jack, lastName=Pep, hireDate=1973-03-01, termDate=null, sales(0)={}
```

2.4. Parameters

- Runtime query parameters passed into query

Figure 2.5. Name-based Query Parameters

```
TypedQuery<Customer> query = em.createQuery(
        "select c from Customer c " +
        "where c.firstName=:firstName and c.lastName=:lastName",
        Customer.class);
query.setParameter("firstName", "cat");
query.setParameter("lastName", "inhat");

Customer customer = query.getSingleResult();
assertNotNull(customer);
```

```
log.info("found customer for param names:" + customer);
```

```
select
    customer0_.CUSTOMER_ID as CUSTOMER1_1_,
    customer0_.FIRST_NAME as FIRST2_1_,
    customer0_.LAST_NAME as LAST3_1_
from JPAQL_CUSTOMER customer0_
where customer0_.FIRST_NAME=? and customer0_.LAST_NAME=?

-found customer for param names:firstName=cat, lastName=inhat
```

- :firstName and :lastName act as placeholders for runtime query parameters
- Runtime parameters supplied using placeholder names
- A parameter for each placeholder must be supplied - no defaults
- A placeholder must exist for each parameter supplied - no extras

Figure 2.6. Ordinal-based Parameters

```
query = em.createQuery(
        "select c from Customer c " +
        "where c.firstName=?1 and c.lastName like ?2", Customer.class);
query.setParameter(1, "thing");
query.setParameter(2, "%");
List<Customer> customers = query.getResultList();
assertTrue("unexpected number of customers:" + customers.size(),
        customers.size() == 2);
for(Customer c : customers) {
    log.info("found customer for param position:" + c);
}
```

```
    select
        customer0_.CUSTOMER_ID as CUSTOMER1_1_,
        customer0_.FIRST_NAME as FIRST2_1_,
        customer0_.LAST_NAME as LAST3_1_
    from JPAQL_CUSTOMER customer0_
    where customer0_.FIRST_NAME=?  and ( customer0_.LAST_NAME like ? )
-found customer for param position:firstName=thing, lastName=one
-found customer for param position:firstName=thing, lastName=two
```

- Appended numbers (?1) assign an ordinal value
- No numbers supplied (?) cause default value based on order

Figure 2.7. Date-based Parameters

```
Calendar hireDate = Calendar.getInstance();
hireDate.set(Calendar.YEAR, 1972);
TypedQuery<Clerk> query = em.createQuery(
        "select c from Clerk c " +
        "where c.hireDate > :date", Clerk.class);
query.setParameter("date", hireDate.getTime(), TemporalType.DATE);
```

```
Clerk clerk = query.getSingleResult();
log.info("found clerk by date(" + hireDate.getTime() + "):" + clerk);
```

```
  select
    clerk0_.CLERK_ID as CLERK1_0_,
    clerk0_.FIRST_NAME as FIRST2_0_,
    clerk0_.HIRE_DATE as HIRE3_0_,
    clerk0_.LAST_NAME as LAST4_0_,
    clerk0_.TERM_DATE as TERM5_0_
  from JPAQL_CLERK clerk0_
  where clerk0_.HIRE_DATE>?
...
-found clerk by date(Fri Oct 06 20:28:08 EDT 1972):firstName=Jack, lastName=Pep, hireDate=1973-03-01,
 termDate=null, sales(0)={}
```

- Dates are specified as DATE, TIME, or TIMESTAMP

2.5. Paging Properties

Figure 2.8.

```
TypedQuery<Sale> query = em.createQuery(
    "select s from Sale s", Sale.class);
for(int i=0; i<2; i++) {
  List<Sale> sales = query.setMaxResults(10)
                          .setFirstResult(i)
                          .getResultList();
  for(Sale s: sales) {
    log.info("found sale in page(" + i + "):" + s);
    em.detach(s); //we are done with this
  }
}
```

```
  select
    sale0_.SALE_ID as SALE1_2_,
    sale0_.amount as amount2_2_,
    sale0_.BUYER_ID as BUYER3_2_,
    sale0_.date as date4_2_,
    sale0_.SALE_STORE as SALE5_2_
  from
    JPAQL_SALE sale0_ limit ?
...
-found sale in page(0):date=1998-04-10 10:13:35, amount=$100.00, buyer=1, clerks(1)={1, }
-found sale in page(0):date=1999-06-11 14:15:10, amount=$150.00, buyer=2, clerks(2)={1, 2, }
  select
    sale0_.SALE_ID as SALE1_2_,
    sale0_.amount as amount2_2_,
    sale0_.BUYER_ID as BUYER3_2_,
    sale0_.date as date4_2_,
    sale0_.SALE_STORE as SALE5_2_
  from
    JPAQL_SALE sale0_ limit ? offset ?
...
```

```
-found sale in page(1):date=1999-06-11 14:15:10, amount=$150.00, buyer=2, clerks(2)={1, 2, }
```

- Offset and limits passed to database
- Database provides specified subset of rows

2.6. Pessimistic Locking

- Obtain a locked copy of entity -- ready for modification
- Required for some concurrent interactions with database

Figure 2.9. Obtaining a Pessimistic Write Lock

```
//get a list of clerks to update -- locked so others cannot change
List<Clerk> clerks = em.createQuery(
        "select c from Clerk c " +
        "where c.hireDate > :date", Clerk.class)
        .setParameter("date", new GregorianCalendar(1972,Calendar.JANUARY,1).getTime())
        .setLockMode(LockModeType.PESSIMISTIC_WRITE)
        .setHint("javax.persistence.lock.timeout", 0)
        .getResultList();
//make changes
for (Clerk c: clerks) {
    c.setHireDate(new GregorianCalendar(1972, Calendar.FEBRUARY, 1).getTime());
}
```

```
select
    clerk0_.CLERK_ID as CLERK1_0_,
    clerk0_.FIRST_NAME as FIRST2_0_,
    clerk0_.HIRE_DATE as HIRE3_0_,
    clerk0_.LAST_NAME as LAST4_0_,
    clerk0_.TERM_DATE as TERM5_0_
from JPAQL_CLERK clerk0_
where clerk0_.HIRE_DATE>?
for update
...
```

- Provider adds database-specific technique for lock
- Lock timeout (in msecs) can be expressed through query hint

> **Note**
>
> Not all databases support lock timeouts

2.7. Bulk Updates

- Change database -- not query it
- Bypasses cache -- cached entities out of sync with database changes
- Criteria API updates/deletes added in JPA 2.1

Figure 2.10. JPA-QL Bulk Update Example

```
Query update = em.createQuery(
      "update Clerk c set c.lastName=:newlast where c.lastName=:last");
update.setParameter("last", "Pep");
update.setParameter("newlast", "Peppy");
int rows = update.executeUpdate();
assertEquals("unexpected rows updated:" + rows, clerks.size(), rows);
```

```
update JPAQL_CLERK
set LAST_NAME=?
where LAST_NAME=?
```

- Change directly applied to database, not the cached entity
- Number of entities changed returned

Figure 2.11. Criteria API Bulk Update Example

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaUpdate<Clerk> qdef2=cb.createCriteriaUpdate(Clerk.class);

//"update Clerk c set c.lastName=:newlast where c.lastName=:last"
Root<Clerk> c2 = qdef2.from(Clerk.class);
qdef2.set("lastName", "Peppy")
    .where(cb.equal(c2.get("lastName"), "Pep"));

Query update = em.createQuery(qdef2);
int rows = update.executeUpdate();
assertEquals("unexpected rows updated:" + rows, clerks.size(), rows);
```

Figure 2.12. JPA-QL Bulk Delete Example

```
Query update = em.createQuery(
      "delete from Customer c " +
      "where c.firstName like :first AND c.lastName like :last");
int rows = update.setParameter("first", "thing")
             .setParameter("last", "%")
             .executeUpdate();
assertTrue("no rows updated", rows > 0);
```

```
delete from JPAQL_CUSTOMER
where ( FIRST_NAME like ? ) and ( LAST_NAME like ? )
```

- Bulk deletes do not trigger cascades
- Entity instance exists in memory even after deleted from database

Figure 2.13. Criteria API Bulk Update Example

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaDelete<Customer> delete = cb.createCriteriaDelete(Customer.class);
```

```
//"delete from Customer c " +
//"where c.firstName like :first AND c.lastName like :last");
Root<Customer> c2 = delete.from(Customer.class);
delete.where(cb.and(
    cb.like(c.<String>get("firstName"), "thing"),
    cb.like(c.<String>get("lastName"), "%")
    ));

Query update = em.createQuery(delete);
int rows = update.executeUpdate();
assertTrue("no rows updated", rows > 0);
```

Figure 2.14. Refresh/Clear/Detach Stale Entit(ies)

```
//re-sync entity with DB changes
em.refresh(clerk);
//evict all managed entities in persistence context
em.clear();
//remove entity from persistence context
em.detach(clerk);
```

- Keeping stale entities around will produce confusing results
- "em.clear()" should be avoided except at end of transaction since un-manages everything

2.8. Named Queries

- Register query with provider rather than ad-hoc
- Available for JPA-QL and Native SQL -- not available with Criteria API
- Can be registered using class annotations and orm.xml descriptor
- LockMode and hints can be specified in declaration

Figure 2.15. Named Query Annotations Applied to (any) Entity Class

```
@Entity
@Table(name="JPAQL_CUSTOMER")
@NamedQueries({
  @NamedQuery(name="Customer.getCustomersByName",
      query="select c from Customer c " +
          "where c.firstName like :first AND c.lastName like :last"),
  @NamedQuery(name="Customer.getCustomerPurchases",
      query="select s from Sale s " +
          "where s.buyerId=:custId")
})
public class Customer {
```

Figure 2.16. Using Named Query

```
Customer customer =
  em.createNamedQuery("Customer.getCustomersByName", Customer.class)
   .setParameter("first", "cat")
```

```
    .setParameter("last", "inhat")
    .getResultList()
    .get(0);
assertNotNull("no customer found", customer);
```

Figure 2.17. Named Native Query Annotation Example

```
@Entity
@Table(name="JPAQL_CUSTOMER")
@NamedNativeQueries({
  @NamedNativeQuery(name="Customer.getCustomerRows",
      query="select * from JPAQL_CUSTOMER c " +
         "where c.FIRST_NAME = ?1")
})
public class Customer {
```

- Example query uses Native SQL to return all columns for table

Figure 2.18. Using Named Native Query

```
@SuppressWarnings("unchecked")
List<Object[]> rows = em.createNamedQuery("Customer.getCustomerRows")
     .setParameter(1, "cat")
     .getResultList();
assertEquals("unexpected customers found", 1, rows.size());
log.info("found customer:" + Arrays.toString(rows.get(0)));
```

```
  select * from JPAQL_CUSTOMER c
  where c.FIRST_NAME = ?
-found customer:[1, cat, inhat]
```

2.9. Summary

- Untyped (JPA 1.0) and Typed (JPA 2.0) Queries
- Single and multiple results
- Named and ordinal parameters
- DATE, TIME, and TIMSTAMP parameters
- Offset(firstResult) and limit(maxResults) paging
- Locking
- JPA-QL and (JPA 2.1) Criteria Bulk Updates
- Named Queries

Part II. JPAQL

JPA Query Language

3.1. Simple Entity Query

Figure 3.1. Example JPA-QL Query

```
select object(c) from Customer as c
```

Figure 3.2. Alternate JPA-QL Query Form

```
select c from Customer c
```

- "select" defines root query objects -- all path references must start from this set
- "from" defines source of root query terms
- "as" (optional) identifies a variable assignment of entity in from clause
- "object()" (optional) identifies what is returned for the path expressed in select clause (e.g., object(), count()) -- left over from EJBQL
- no "where" clause indicates all entities are selected

Figure 3.3. Using a JPA-QL Query

```
TypedQuery<Customer> query = em.createQuery(
        "select object(c) from Customer as c",
        Customer.class);
List<Customer> results = query.getResultList();
```

```
    select
        customer0_.CUSTOMER_ID as CUSTOMER1_3_,
        customer0_.FIRST_NAME as FIRST2_3_,
        customer0_.LAST_NAME as LAST3_3_
    from JPAQL_CUSTOMER customer0_
 -found result:firstName=cat, lastName=inhat
 -found result:firstName=thing, lastName=one
 -found result:firstName=thing, lastName=two
```

3.2. Non-Entity Queries

Figure 3.4. Non-Entity Query Example

```
select c.lastName from Customer c
```

- Allows return of simple property
- "c.lastName" is called a "path"
- All paths based from root query terms

- Single path selects return typed list of values

Figure 3.5. Using Non-Entity Query

```
TypedQuery<String> query = em.createQuery(
        "select c.lastName from Customer c", String.class);
List<String> results = query.getResultList();
```

```
    select customer0_.LAST_NAME as col_0_0_
    from JPAQL_CUSTOMER customer0_
 -lastName=inhat
 -lastName=one
 -lastName=two
```

- Query result is a List<String> because "c.lastName" is a String

3.3. Multi-select Query

3.3.1. Multi-select Query with Object[]

Figure 3.6. Multi-select Query with Object[] Example

```
select c.firstName, c.hireDate from Clerk c
```

- Select specifies multiple terms
- Terms are expressed thru a path expression
- Terms must be based off paths from root terms in the FROM (or JOIN) clause

Figure 3.7. Using Object[] Multi-select Query

```
TypedQuery<Object[]> query = em.createQuery(
        "select c.firstName, c.hireDate from Clerk c", Object[].class);
List<Object[]> results = query.getResultList();
assertTrue("no results", results.size() > 0);
for(Object[] result : results) {
    assertEquals("unexpected result length", 2, result.length);
    String firstName = (String) result[0];
    Date hireDate = (Date) result[1];
    log.info("firstName=" + firstName + " hireDate=" + hireDate);
}
```

```
    select
        clerk0_.FIRST_NAME as col_0_0_,
        clerk0_.HIRE_DATE as col_1_0_
    from JPAQL_CLERK clerk0_
 -firstName=Manny hireDate=1970-01-01
 -firstName=Moe hireDate=1970-01-01
 -firstName=Jack hireDate=1973-03-01
```

- Query defined to return elements of select in Object[]

3.3.2. Multi-select Query with Tuple

Figure 3.8. Multi-select Query with Tuple Example

```
select c.firstName as firstName, c.hireDate as hireDate from Clerk c
```

- Aliases may be assigned to select terms for named-access to results

Figure 3.9. Using Tuple Multi-select Query

```
TypedQuery<Tuple> query = em.createQuery(
        "select c.firstName as firstName, c.hireDate as hireDate from Clerk c", Tuple.class);
List<Tuple> results = query.getResultList();
assertTrue("no results", results.size() > 0);
for(Tuple result : results) {
    assertEquals("unexpected result length", 2, result.getElements().size());
    String firstName = result.get("firstName", String.class);
    Date hireDate = result.get("hireDate", Date.class);
    log.info("firstName=" + firstName + " hireDate=" + hireDate);
}
```

```
    select
        clerk0_.FIRST_NAME as col_0_0_,
        clerk0_.HIRE_DATE as col_1_0_
    from JPAQL_CLERK clerk0_
 -firstName=Manny hireDate=1970-01-01
 -firstName=Moe hireDate=1970-01-01
 -firstName=Jack hireDate=1973-03-01
```

- Query defined to return instances of Tuple class
- Tuples provide access using
    - get(index) - simular to Object[]
    - get(index, Class<T> resultType) - typed access by index
    - get(alias) - access by alias
    - get(alias, Class<T> resultType) - typed access by alias
    - getElements() - access thru collection interface

3.3.3. Multi-select Query with Constructor

Figure 3.10. Multi-select Query with Constructor Example

```
select new ejava.jpa.examples.query.Receipt(s.id, s.buyerId, s.date, s.amount)
from Sale s
```

- Individual elements of select are matched up against class constructor

Figure 3.11. Example ResultClass

```
package ejava.jpa.examples.query;
```

```
...
public class Receipt {
    private long saleId;
    private long customerId;
    private Date date;
    private double amount;

    public Receipt(long saleId, long customerId, Date date, BigDecimal amount) {
        this(customerId, saleId, date, amount.doubleValue());
    }
    public Receipt(long saleId, long customerId, Date date, double amount) {
        this.customerId = customerId;
        this.saleId = saleId;
        this.date = date;
        this.amount = amount;
    }
...
```

- Constructed class may be simple POJO -- no need to be an entity
- Instances are not managed
- Suitable for use as Data Transfer Objects (DTOs)

Figure 3.12. Using Constructor Multi-select Query

```
TypedQuery<Receipt> query = em.createQuery(
    String.format("select new %s(", Receipt.class.getName()) +
    "s.id,s.buyerId,s.date, s.amount) " +
    "from Sale s", Receipt.class);

List<Receipt> results = query.getResultList();
for(Receipt receipt : results) {
    assertNotNull("no receipt", receipt);
    log.info("receipt=" + receipt);
}
```

```
    select
        sale0_.SALE_ID as col_0_0_,
        sale0_.BUYER_ID as col_1_0_,
        sale0_.date as col_2_0_,
        sale0_.amount as col_3_0_
    from JPAQL_SALE sale0_
 -receipt=sale=1, customer=1, date=1998-04-10 10:13:35, amount=$100.00
 -receipt=sale=2, customer=2, date=1999-06-11 14:15:10, amount=$150.00
```

- Each row returned as instance of provided class

3.4. Path Expressions

3.4.1. Single Element Path Expressions

Figure 3.13. Example Single Element Path Query

```
select s.id, s.store.name from Sale s
```

- All paths based off root-level FROM (or JOIN) terms
- Paths use dot (".") notation to change contexts
- Paths -- used this way -- must always express a single element. Must use JOINs for paths involving collections
- Paths that cross entity boundaries automatically add a join to SQL query

Figure 3.14. Using Single Element Path Expression

```
TypedQuery<Object[]> query = em.createQuery(
        "select s.id, s.store.name from Sale s", Object[].class);
List<Object[]> results = query.getResultList();
assertTrue("no results", results.size() > 0);
for(Object[] result : results) {
    assertEquals("unexpected result length", 2, result.length);
    Long id = (Long) result[0];
    String name = (String) result[1];
    log.info("sale.id=" + id + ", sale.store.name=" + name);
}
```

```
    select
        sale0_.SALE_ID as col_0_0_,
        store1_.name as col_1_0_
    from JPAQL_SALE sale0_,
         ORMQL_STORE store1_
    where sale0_.SALE_STORE=store1_.STORE_ID

 -sale.id=1, sale.store.name=Big Al's
 -sale.id=2, sale.store.name=Big Al's
```

- Automatic INNER JOIN formed between Sale and Store because of the cross-entity path

3.4.2. Collection Element Path Expressions

3.4.2.1. INNER JOIN Collection Path Expressions

Figure 3.15. Illegal Collection Path Expression

```
select c.sales.date from Clerk c
```

- Cannot directly navigate a XxxToMany relationship without a join

Figure 3.16. Correct Collection Path Expression

```
select sale.date from Clerk c INNER JOIN c.sales sale
```

- Collection ("sales") is brought in as a root term ("sale") of the query through a JOIN expression
- JOINs will match entities by their defined primary/foreign keys
- INNER JOIN will return only those entities where there is a match

Figure 3.17. Alternate Collection Path Expression

```
select sale.date from Clerk c JOIN c.sales sale
```

- INNER JOIN is the default

Figure 3.18. Alternate EJB-QL Form

```
select sale.date from Clerk c, IN (c.sales) sale
```

Figure 3.19. Collection Path Expression SQL Output

```
    select sale2_.date as col_0_0_
    from JPAQL_CLERK clerk0_
    inner join JPAQL_SALE_CLERK_LINK sales1_
        on clerk0_.CLERK_ID=sales1_.CLERK_ID
    inner join JPAQL_SALE sale2_
        on sales1_.SALE_ID=sale2_.SALE_ID

 -found result:1998-04-10 10:13:35.0
 -found result:1999-06-11 14:15:10.0
 -found result:1999-06-11 14:15:10.0
```

- (Many-to-Many) Link table used during JOIN
- Tables automatically joined on primary keys
- Only Sales sold by our Clerks are returned

3.4.2.2. LEFT OUTER JOIN Collection Path Expressions

Figure 3.20. LEFT OUTER JOIN Example

```
select c.id, c.firstName, sale.amount
from Clerk c
LEFT OUTER JOIN c.sales sale
```

- LEFT is the default for OUTER JOIN

Figure 3.21. Alternate LEFT OUTER JOIN Form

```
select c.id, c.firstName, sale.amount
```

```
from Clerk c
LEFT JOIN c.sales sale
```

- LEFT OUTER JOIN will return root with or without related entities

Figure 3.22. LEFT OUTER JOIN Runtime SQL Output

```
  select
    clerk0_.CLERK_ID as col_0_0_,
    clerk0_.FIRST_NAME as col_1_0_,
    sale2_.amount as col_2_0_
  from JPAQL_CLERK clerk0_
  left outer join JPAQL_SALE_CLERK_LINK sales1_
      on clerk0_.CLERK_ID=sales1_.CLERK_ID
  left outer join JPAQL_SALE sale2_
      on sales1_.SALE_ID=sale2_.SALE_ID

-clerk.id=1, clerk.firstName=Manny, amount=100.00
-clerk.id=1, clerk.firstName=Manny, amount=150.00
-clerk.id=2, clerk.firstName=Moe, amount=150.00
-clerk.id=3, clerk.firstName=Jack, amount=null
```

- (Many-to-Many) Link table used during JOIN
- Tables automatically joined on primary keys
- All clerks, with or without a Sale, are returned

3.4.2.3. Explicit Collection Path Expressions

Figure 3.23. Explicit Collection Path Example

```
select c from Sale s, Customer c where c.id = s.buyerId
```

- Permits JOINs without relationship in entity model

Figure 3.24. Explicit Collection Path SQL Output

```
  select
    customer1_.CUSTOMER_ID as CUSTOMER1_3_,
    customer1_.FIRST_NAME as FIRST2_3_,
    customer1_.LAST_NAME as LAST3_3_
  from JPAQL_SALE sale0_ cross
  join JPAQL_CUSTOMER customer1_
  where customer1_.CUSTOMER_ID=sale0_.BUYER_ID

-found result:firstName=cat, lastName=inhat
-found result:firstName=thing, lastName=one
```

- Returns all Customers that are identified by a Sale

3.5. Eager Fetching through JOINs

3.5.1. Lazy Fetch Problem

Figure 3.25. Example Query Resulting in Lazy Fetch

```
select s from Store s JOIN s.sales
where s.name='Big Al''s'
```

- A normal JOIN (implicit or explicit) may honor the fetch=LAZY property setting of the relation
- Can be exactly what is desired
- Can also cause problems or extra work if not desired

Figure 3.26. Example Entity with Lazy Fetch Declared for Relation

```
@Entity @Table(name="ORMQL_STORE")
public class Store {
...
    @OneToMany(mappedBy="store",
            cascade={CascadeType.REMOVE},
            fetch=FetchType.LAZY)
    private List<Sale> sales = new ArrayList<Sale>();
```

- Sales are lazily fetched when obtaining Store

Figure 3.27. Example Lazy Fetch Problem

```
Store store = em2.createQuery(
        "select s from Store s JOIN s.sales " +
        "where s.name='Big Al''s'",
        Store.class).getSingleResult();
em2.close();
try {
    store.getSales().get(0).getAmount();
    fail("did not trigger lazy initialization exception");
} catch (LazyInitializationException expected) {
    log.info("caught expected exception:" + expected);
}
```

```
    select
        store0_.STORE_ID as STORE1_0_,
        store0_.name as name0_
    from ORMQL_STORE store0_
    inner join JPAQL_SALE sales1_
            on store0_.STORE_ID=sales1_.SALE_STORE
    where store0_.name='Big Al''s' limit ?

 -caught expected exception:org.hibernate.LazyInitializationException:
     failed to lazily initialize a collection of role:
     ejava.jpa.examples.query.Store.sales, no session or session was closed
```

- Accessing the Sale properties causes a LazyInitializationException when persistence context no longer active or accessible

> ### One Row per Parent is Returned for fetch=LAZY
>
> Note that only a single row is required to be returned from the database for a fetch=LAZY relation. Although it requires more queries to the database, it eliminates duplicate parent information for each child row and can eliminate the follow-on query all together when not accessed.

3.5.2. Adding Eager Fetch during Query

Figure 3.28. Example Eager Fetch Query

```
select s from Store s JOIN FETCH s.sales
where s.name='Big Al''s'
```

- A JOIN FETCH used to eager load related entities as side-effect of query
- Can be used as substitute for fetch=EAGER specification on relation

Figure 3.29. Example Eager Fetch SQL Output

```
  select
    store0_.STORE_ID as STORE1_0_0_,
    sales1_.SALE_ID as SALE1_1_1_,
    store0_.name as name0_0_,
    sales1_.amount as amount1_1_,
    sales1_.BUYER_ID as BUYER3_1_1_,
    sales1_.date as date1_1_,
    sales1_.SALE_STORE as SALE5_1_1_,
    sales1_.SALE_STORE as SALE5_0_0__,
    sales1_.SALE_ID as SALE1_0__
  from ORMQL_STORE store0_
  inner join JPAQL_SALE sales1_ on store0_.STORE_ID=sales1_.SALE_STORE
  where store0_.name='Big Al''s'
```

- Sales are eagerly fetched when obtaining Store

> ### Parent Rows Repeated for each Child for fetch=EAGER
>
> Note that adding JOIN FETCH to parent query causes the parent rows to be repeated for each eagerly loaded child row and eliminated by the provider. This requires fewer database queries but results in more (and redundant) data to be returned from the query.

3.6. Distinct Results

Figure 3.30. Distinct Example

```
select DISTINCT c.lastName from Customer c
```

- Limits output to unique value combinations

Figure 3.31. Distinct Example Output

```
    select
        distinct customer0_.LAST_NAME as col_0_0_
    from JPAQL_CUSTOMER customer0_
 -found result:two
 -found result:inhat
 -found result:one
```

- Found three unique last names

Figure 3.32. Another Distinct Example

```
select DISTINCT c.firstName from Customer c
```

Figure 3.33. Another Distinct Example Output

```
    select
        distinct customer0_.FIRST_NAME as col_0_0_
    from JPAQL_CUSTOMER customer0_
 -found result:cat
 -found result:thing
```

- Found two unique first names

3.7. Summary

- Element queries return property data without managed entities
- Multi-slect element query return types (Object[], Tuple, ResultClass)
- Path expressions
  - Single element
  - Collection element (JOIN)
  - INNER and OUTER JOIN
- Eager loading with Join Fetch

JPAQL Where Clauses

4.1. Equality Test

Figure 4.1. Example Equality Test

```
select c from Customer c
where c.firstName='cat'
```

```
    select
        customer0_.CUSTOMER_ID as CUSTOMER1_3_,
        customer0_.FIRST_NAME as FIRST2_3_,
        customer0_.LAST_NAME as LAST3_3_
    from JPAQL_CUSTOMER customer0_
    where customer0_.FIRST_NAME='cat'

 -found result:firstName=cat, lastName=inhat
```

- Return entities where there is an equality match

Figure 4.2. Escaping Special Characters

```
select s from Store s
where s.name='Big Al''s'
```

```
    select
        store0_.STORE_ID as STORE1_0_,
        store0_.name as name0_
    from ORMQL_STORE store0_
    where store0_.name='Big Al''s'
...
 -found result:name=Big Al's, sales(2)={1, 2, }
```

- Escaped special character is passed through to the database

4.2. Like Test

Figure 4.3. Like Test Literal

```
select c from Clerk c
where c.firstName like 'M%'
```

```
    select
        clerk0_.CLERK_ID as CLERK1_2_,
        clerk0_.FIRST_NAME as FIRST2_2_,
        clerk0_.HIRE_DATE as HIRE3_2_,
        clerk0_.LAST_NAME as LAST4_2_,
        clerk0_.TERM_DATE as TERM5_2_
    from JPAQL_CLERK clerk0_
    where clerk0_.FIRST_NAME like 'M%'
...
 -found result:firstName=Manny, lastName=Pep, hireDate=1970-01-01, termDate=null, sales(2)={1, 2, }
 -found result:firstName=Moe, lastName=Pep, hireDate=1970-01-01, termDate=null, sales(1)={2, }
```

Figure 4.4. Like Test Literal Parameter

```
select c from Clerk c
where c.firstName like :firstName)
```

Figure 4.5. Using Like Test Literal Parameter

```
TypedQuery<T> query = em.createQuery(ejbqlString, resultType);
query.setParameter("firstName", "M%");
List<T> objects = query.getResultList();
```

```
    select
        clerk0_.CLERK_ID as CLERK1_2_,
        clerk0_.FIRST_NAME as FIRST2_2_,
        clerk0_.HIRE_DATE as HIRE3_2_,
        clerk0_.LAST_NAME as LAST4_2_,
        clerk0_.TERM_DATE as TERM5_2_
    from JPAQL_CLERK clerk0_
    where clerk0_.FIRST_NAME like ?

 -found result:firstName=Manny, lastName=Pep, hireDate=1970-01-01, termDate=null, sales(2)={1, 2, }
 -found result:firstName=Moe, lastName=Pep, hireDate=1970-01-01, termDate=null, sales(1)={2, }
```

Figure 4.6. Like Test Concatenated String

```
select c from Clerk c
where c.firstName like concat(:firstName,'%')
```

Figure 4.7. Using Like Test Concatenated String

```
TypedQuery<T> query = em.createQuery(ejbqlString, resultType);
query.setParameter("firstName", "M");
List<T> objects = query.getResultList();
```

```
    select
        clerk0_.CLERK_ID as CLERK1_2_,
        clerk0_.FIRST_NAME as FIRST2_2_,
        clerk0_.HIRE_DATE as HIRE3_2_,
        clerk0_.LAST_NAME as LAST4_2_,
        clerk0_.TERM_DATE as TERM5_2_
    from JPAQL_CLERK clerk0_
    where clerk0_.FIRST_NAME like (?||'%')

 -found result:firstName=Manny, lastName=Pep, hireDate=1970-01-01, termDate=null, sales(2)={1, 2, }
 -found result:firstName=Moe, lastName=Pep, hireDate=1970-01-01, termDate=null, sales(1)={2, }
```

Figure 4.8. Like Test Single Character Wildcard

```
select c from Clerk c
where c.firstName like '_anny'
```

```
    select
        clerk0_.CLERK_ID as CLERK1_2_,
        clerk0_.FIRST_NAME as FIRST2_2_,
        clerk0_.HIRE_DATE as HIRE3_2_,
        clerk0_.LAST_NAME as LAST4_2_,
        clerk0_.TERM_DATE as TERM5_2_
    from
        JPAQL_CLERK clerk0_
    where
        clerk0_.FIRST_NAME like '_anny'

 -found result:firstName=Manny, lastName=Pep, hireDate=1970-01-01, termDate=null, sales(2)={1, 2, }
```

4.3. Formulas

Figure 4.9. Example Formula

```
select s from Sale s
where (s.amount * :tax) > :amount
```

Figure 4.10. Using Formula

```
String jpaql = "select count(s) from Sale s " +
    "where (s.amount * :tax) > :amount";
TypedQuery<Number> query = em.createQuery(jpaql, Number.class)
        .setParameter("amount", new BigDecimal(10.00));

//keep raising taxes until somebody pays $10.00 in tax
double tax = 0.05;
for (;query.setParameter("tax", new BigDecimal(tax))
           .getSingleResult().intValue()==0;
      tax += 0.01) {
    log.debug("tax=" + NumberFormat.getPercentInstance().format(tax));
}
log.info("raise taxes to: " + NumberFormat.getPercentInstance().format(tax));
```

```
    select count(sale0_.SALE_ID) as col_0_0_
    from JPAQL_SALE sale0_
    where sale0_.amount*?>? limit ?
 -tax=5%
    select count(sale0_.SALE_ID) as col_0_0_
    from JPAQL_SALE sale0_
    where sale0_.amount*?>? limit ?
 -tax=6%
```

```
    select count(sale0_.SALE_ID) as col_0_0_
    from JPAQL_SALE sale0_
    where sale0_.amount*?>? limit ?
 -raise taxes to: 7%
```

4.4. Logic Operators

Figure 4.11. Logic Operator Example

```
select c from Customer c
where (c.firstName='cat' AND c.lastName='inhat')
    OR c.firstName='thing'
```

Figure 4.12. Logic Operator Example Output

```
    select
        customer0_.CUSTOMER_ID as CUSTOMER1_3_,
        customer0_.FIRST_NAME as FIRST2_3_,
        customer0_.LAST_NAME as LAST3_3_
    from JPAQL_CUSTOMER customer0_
    where
        customer0_.FIRST_NAME='cat'
        and customer0_.LAST_NAME='inhat'
        or customer0_.FIRST_NAME='thing'

 -found result:firstName=cat, lastName=inhat
 -found result:firstName=thing, lastName=one
 -found result:firstName=thing, lastName=two
```

Figure 4.13. Another Logic Operator Example

```
select c from Customer c
where (NOT (c.firstName='cat' AND c.lastName='inhat'))
    OR c.firstName='thing'
```

Figure 4.14. Another Logic Operator Example Output

```
    select
        customer0_.CUSTOMER_ID as CUSTOMER1_3_,
        customer0_.FIRST_NAME as FIRST2_3_,
        customer0_.LAST_NAME as LAST3_3_
    from JPAQL_CUSTOMER customer0_
    where
        customer0_.FIRST_NAME<>'cat'
        or customer0_.LAST_NAME<>'inhat'
        or customer0_.FIRST_NAME='thing'

 -found result:firstName=thing, lastName=one
 -found result:firstName=thing, lastName=two
```

4.5. Equality Tests

- Must compare values
  - Of same type
  - Of legal promotion type
    - Can compare 123:int to 123:long
    - Cannot compare 123:int to "123":string
  - Can compare entities

Figure 4.15. Example Entity Equality Query

```
select s from Sale s
  JOIN s.clerks c
  where c = :clerk
```

- Compare entities and not primary/foreign key values

Figure 4.16. Using Entity Equality Query

```
//get a clerk entity
Clerk clerk = em.createQuery(
        "select c from Clerk c where c.firstName = 'Manny'",
        Clerk.class)
       .getSingleResult();

//find all sales that involve this clerk
List<Sale> sales = em.createQuery(
    "select s from Sale s " +
    "JOIN s.clerks c " +
    "where c = :clerk",
    Sale.class)
        .setParameter("clerk", clerk)
        .getResultList();
```

```
    select
        clerk0_.CLERK_ID as CLERK1_2_,
        clerk0_.FIRST_NAME as FIRST2_2_,
        clerk0_.HIRE_DATE as HIRE3_2_,
        clerk0_.LAST_NAME as LAST4_2_,
        clerk0_.TERM_DATE as TERM5_2_
    from JPAQL_CLERK clerk0_
    where clerk0_.FIRST_NAME='Manny' limit ?
```

```
    select
        sale0_.SALE_ID as SALE1_1_,
        sale0_.amount as amount1_,
        sale0_.BUYER_ID as BUYER3_1_,
        sale0_.date as date1_,
        sale0_.SALE_STORE as SALE5_1_
    from JPAQL_SALE sale0_
    inner join JPAQL_SALE_CLERK_LINK clerks1_
```

```
            on sale0_.SALE_ID=clerks1_.SALE_ID
    inner join JPAQL_CLERK clerk2_
            on clerks1_.CLERK_ID=clerk2_.CLERK_ID
    where clerk2_.CLERK_ID=?
...
 -found=date=1998-04-10 10:13:35, amount=$100.00, buyer=1, clerks(1)={1, }
...
 -found=date=1999-06-11 14:15:10, amount=$150.00, buyer=2, clerks(2)={1, 2, }
```

4.6. Between

Figure 4.17. Example Between Query

```
select s from Sale s
where s.amount BETWEEN :low AND :high
```

```
    select
        sale0_.SALE_ID as SALE1_1_,
        sale0_.amount as amount1_,
        sale0_.BUYER_ID as BUYER3_1_,
        sale0_.date as date1_,
        sale0_.SALE_STORE as SALE5_1_
    from
        JPAQL_SALE sale0_
    where
        sale0_.amount between ? and ?
...
 -found result:date=1998-04-10 10:13:35, amount=$100.00, buyer=1, clerks(1)={1, }
```

Figure 4.18. Another Example Between Query

```
select s from Sale s
where s.amount NOT BETWEEN :low AND :high
```

```
    select
        sale0_.SALE_ID as SALE1_1_,
        sale0_.amount as amount1_,
        sale0_.BUYER_ID as BUYER3_1_,
        sale0_.date as date1_,
        sale0_.SALE_STORE as SALE5_1_
    from JPAQL_SALE sale0_
    where sale0_.amount not between ? and ?
...
 -found result:date=1999-06-11 14:15:10, amount=$150.00, buyer=2, clerks(2)={1, 2, }
```

4.7. Testing for Null

Can be used to test for unassigned value or relationship

Figure 4.19. Example Test for Null

```
select s from Sale s
where s.store IS NULL
```

```
  select
    sale0_.SALE_ID as SALE1_1_,
    sale0_.amount as amount1_,
    sale0_.BUYER_ID as BUYER3_1_,
    sale0_.date as date1_,
    sale0_.SALE_STORE as SALE5_1_
  from
    JPAQL_SALE sale0_
  where
    sale0_.SALE_STORE is null
```

Figure 4.20. Example Test for Not Null

```
select s from Sale s
where s.store IS NOT NULL
```

```
  select
    sale0_.SALE_ID as SALE1_1_,
    sale0_.amount as amount1_,
    sale0_.BUYER_ID as BUYER3_1_,
    sale0_.date as date1_,
    sale0_.SALE_STORE as SALE5_1_
  from JPAQL_SALE sale0_
  where sale0_.SALE_STORE is not null
...
 -found result:date=1998-04-10 10:13:35, amount=$100.00, buyer=1, clerks(1)={1, }
...
 -found result:date=1999-06-11 14:15:10, amount=$150.00, buyer=2, clerks(2)={1, 2, }
```

4.8. Testing Empty Collection

Can be used to test for an empty collection

Figure 4.21. Example Empty Collection Test

```
select c from Clerk c
where c.sales IS EMPTY
```

```
  select
    clerk0_.CLERK_ID as CLERK1_2_,
    clerk0_.FIRST_NAME as FIRST2_2_,
    clerk0_.HIRE_DATE as HIRE3_2_,
    clerk0_.LAST_NAME as LAST4_2_,
    clerk0_.TERM_DATE as TERM5_2_
  from JPAQL_CLERK clerk0_
  where not (exists (
        select sale2_.SALE_ID from JPAQL_SALE_CLERK_LINK sales1_, JPAQL_SALE sale2_
        where clerk0_.CLERK_ID=sales1_.CLERK_ID and sales1_.SALE_ID=sale2_.SALE_ID))
...
 -found result:firstName=Jack, lastName=Pep, hireDate=1973-03-01, termDate=null, sales(0)={}
```

- Sub-select returns values from collection under test
- Outer query tests for no existing (EMPTY)values

Figure 4.22. Example Non-Empty Test

```
select c from Clerk c
where c.sales IS NOT EMPTY
```

```
    select
        clerk0_.CLERK_ID as CLERK1_2_,
        clerk0_.FIRST_NAME as FIRST2_2_,
        clerk0_.HIRE_DATE as HIRE3_2_,
        clerk0_.LAST_NAME as LAST4_2_,
        clerk0_.TERM_DATE as TERM5_2_
    from JPAQL_CLERK clerk0_
    where exists (
            select sale2_.SALE_ID
            from JPAQL_SALE_CLERK_LINK sales1_, JPAQL_SALE sale2_
            where clerk0_.CLERK_ID=sales1_.CLERK_ID and sales1_.SALE_ID=sale2_.SALE_ID
        )
...
 -found result:firstName=Manny, lastName=Pep, hireDate=1970-01-01, termDate=null, sales(2)={1, 2, }
...
 -found result:firstName=Moe, lastName=Pep, hireDate=1970-01-01, termDate=null, sales(1)={2, }
```

- Sub-select returns values from collection under test
- Outer query tests for existing (NOT EMPTY)values

4.9. Membership Test

Can be used to determine membership in a collection

Figure 4.23. Example Membership Test

```
select c from Clerk c
where c.firstName = 'Manny'
```

```
select s from Sale s
where :clerk MEMBER OF s.clerks
```

- Defines a shorthand for a subquery

Figure 4.24. Using Membership Test

```
//get a clerk entity
```

```
Clerk clerk = em.createQuery(
  "select c from Clerk c where c.firstName = 'Manny'",
  Clerk.class)
   .getSingleResult();

//find all sales that involve this clerk
List<Sale> sales = em.createQuery(
  "select s from Sale s " +
  "where :clerk MEMBER OF s.clerks",
  Sale.class)
    .setParameter("clerk", clerk)
    .getResultList();
```

Figure 4.25. Using Membership Test Runtime Output

```
  select
    clerk0_.CLERK_ID as CLERK1_2_,
    clerk0_.FIRST_NAME as FIRST2_2_,
    clerk0_.HIRE_DATE as HIRE3_2_,
    clerk0_.LAST_NAME as LAST4_2_,
    clerk0_.TERM_DATE as TERM5_2_
  from JPAQL_CLERK clerk0_
  where clerk0_.FIRST_NAME='Manny' limit ?
```

```
  select
    sale0_.SALE_ID as SALE1_1_,
    sale0_.amount as amount1_,
    sale0_.BUYER_ID as BUYER3_1_,
    sale0_.date as date1_,
    sale0_.SALE_STORE as SALE5_1_
  from JPAQL_SALE sale0_
  where
    ? in (
      select
        clerk2_.CLERK_ID
      from
        JPAQL_SALE_CLERK_LINK clerks1_,
        JPAQL_CLERK clerk2_
      where
        sale0_.SALE_ID=clerks1_.SALE_ID
        and clerks1_.CLERK_ID=clerk2_.CLERK_ID
    )
...
 -found=date=1998-04-10 10:13:35, amount=$100.00, buyer=1, clerks(1)={1, }
...
 -found=date=1999-06-11 14:15:10, amount=$150.00, buyer=2, clerks(2)={1, 2, }
```

4.10. Subqueries

Useful when query cannot be expressed through JOINs

Figure 4.26. Example Subquery

```
select c from Customer c
where c.id IN
  (select s.buyerId from Sale s
   where s.amount > 100)
```

Figure 4.27. Example Subquery Runtime Output

```
  select
    customer0_.CUSTOMER_ID as CUSTOMER1_3_,
    customer0_.FIRST_NAME as FIRST2_3_,
    customer0_.LAST_NAME as LAST3_3_
  from JPAQL_CUSTOMER customer0_
  where customer0_.CUSTOMER_ID in (
      select sale1_.BUYER_ID
      from JPAQL_SALE sale1_
      where sale1_.amount>100
    )
 -found result:firstName=thing, lastName=one
```

4.11. All

- All existing values must meet criteria (i.e., no value may fail criteria)
- Zero values is the lack of failure (i.e., meets criteria)

Figure 4.28. Example ALL Query

```
select c from Clerk c
where 125 < ALL
(select s.amount from c.sales s)
```

- List all clerks that have all sales above $125.00 or none at all
- -or- List all clerks with no sale <= $125.00

Figure 4.29. Example ALL Query Runtime Output

```
  select
    clerk0_.CLERK_ID as CLERK1_2_,
    clerk0_.FIRST_NAME as FIRST2_2_,
    clerk0_.HIRE_DATE as HIRE3_2_,
    clerk0_.LAST_NAME as LAST4_2_,
    clerk0_.TERM_DATE as TERM5_2_
  from JPAQL_CLERK clerk0_
  where 125<all (
      select sale2_.amount
      from JPAQL_SALE_CLERK_LINK sales1_, JPAQL_SALE sale2_
      where clerk0_.CLERK_ID=sales1_.CLERK_ID and sales1_.SALE_ID=sale2_.SALE_ID
    )
...
 -found result:firstName=Moe, lastName=Pep, hireDate=1970-01-01, termDate=null, sales(1)={2, }
```

```
...
 -found result:firstName=Jack, lastName=Pep, hireDate=1973-03-01, termDate=null, sales(0)={}
```

- Manny excluded because has 1 sale below $125.00
- Moe included because has only $150.00 sale
- Jack included because has no sales that fail criteria

Figure 4.30. Another ALL Query Example

```
select c from Clerk c
where 125 > ALL
(select s.amount from c.sales s)
```

- List all clerks that have all sales below $125.00 or none at all
- -or- List all clerks with no sale >= $125.00

Figure 4.31. Another ALL Query Example Runtime Output

```
    select
        clerk0_.CLERK_ID as CLERK1_2_,
        clerk0_.FIRST_NAME as FIRST2_2_,
        clerk0_.HIRE_DATE as HIRE3_2_,
        clerk0_.LAST_NAME as LAST4_2_,
        clerk0_.TERM_DATE as TERM5_2_
    from JPAQL_CLERK clerk0_
    where 125>all (
            select sale2_.amount
            from JPAQL_SALE_CLERK_LINK sales1_, JPAQL_SALE sale2_
            where clerk0_.CLERK_ID=sales1_.CLERK_ID  and sales1_.SALE_ID=sale2_.SALE_ID
        )

 -found result:firstName=Jack, lastName=Pep, hireDate=1973-03-01, termDate=null, sales(0)={}
```

- Manny excluded because has 1 sale above $125.00
- Moe excluded because has only $150.00 sale
- Jack included because has no sales that fail criteria

4.12. Any

- Any matching value meets criteria (i.e., one match and you are in)
- Zero values fails to meet the criteria (i.e., must have at least one matching value)

Figure 4.32. Example ANY Query

```
select c from Clerk c
where 125 < ANY
(select s.amount from c.sales s)
```

- List all clerks that have at least one sale above $125.00

Figure 4.33. Example ANY Query Runtime Output

```
 -executing query:select c from Clerk c where 125 < ANY    (select s.amount from c.sales s)
   select
       clerk0_.CLERK_ID as CLERK1_2_,
       clerk0_.FIRST_NAME as FIRST2_2_,
       clerk0_.HIRE_DATE as HIRE3_2_,
       clerk0_.LAST_NAME as LAST4_2_,
       clerk0_.TERM_DATE as TERM5_2_
   from JPAQL_CLERK clerk0_
   where 125<any (
           select sale2_.amount
           from JPAQL_SALE_CLERK_LINK sales1_, JPAQL_SALE sale2_
           where clerk0_.CLERK_ID=sales1_.CLERK_ID  and sales1_.SALE_ID=sale2_.SALE_ID
       )
...
 -found result:firstName=Manny, lastName=Pep, hireDate=1970-01-01, termDate=null, sales(2)={1, 2, }
...
 -found result:firstName=Moe, lastName=Pep, hireDate=1970-01-01, termDate=null, sales(1)={2, }
```

- Manny included because has 1 sale above $125.00
- Moe included because $150.00 sale qualifies him as well
- Jack excluded because has no sales that meet criteria

Figure 4.34. Another Example ANY Query

```
select c from Clerk c
where 125 > ANY
(select s.amount from c.sales s)
```

- List all clerks that have at least one sale below $125.00

Figure 4.35. Another Example ANY Query Runtime Output

```
 -executing query:select c from Clerk c where 125 > ANY    (select s.amount from c.sales s)
   select
       clerk0_.CLERK_ID as CLERK1_2_,
       clerk0_.FIRST_NAME as FIRST2_2_,
       clerk0_.HIRE_DATE as HIRE3_2_,
       clerk0_.LAST_NAME as LAST4_2_,
       clerk0_.TERM_DATE as TERM5_2_
   from JPAQL_CLERK clerk0_
   where 125>any (
           select sale2_.amount
           from JPAQL_SALE_CLERK_LINK sales1_, JPAQL_SALE sale2_
           where clerk0_.CLERK_ID=sales1_.CLERK_ID  and sales1_.SALE_ID=sale2_.SALE_ID
       )

 -found result:firstName=Manny, lastName=Pep, hireDate=1970-01-01, termDate=null, sales(2)={1, 2, }
```

- Manny included because has 1 sale below $125.00

- Moe excluded because his only $150.00 sale above criteria
- Jack excluded because has no sales that meet criteria

4.13. Summary

- JPA-QL has detailed coverage of most query needs

Chapter 5.

JPAQL Functions

5.1. String Functions

5.1.1. Base Query

Figure 5.1. Example String Compare

```
select c from Customer c where c.firstName='CAT'
```

```
  select
    customer0_.CUSTOMER_ID as CUSTOMER1_3_,
    customer0_.FIRST_NAME as FIRST2_3_,
    customer0_.LAST_NAME as LAST3_3_
  from JPAQL_CUSTOMER customer0_
  where customer0_.FIRST_NAME='CAT'
```

- No rows found because 'CAT' does not match anything because of case

5.1.2. LOWER

Figure 5.2. Example LOWER Function

```
select c from Customer c where c.firstName=LOWER('CAT')
```

```
  select
    customer0_.CUSTOMER_ID as CUSTOMER1_3_,
    customer0_.FIRST_NAME as FIRST2_3_,
    customer0_.LAST_NAME as LAST3_3_
  from JPAQL_CUSTOMER customer0_
  where customer0_.FIRST_NAME=lower('CAT')

 -found result:firstName=cat, lastName=inhat
```

- One customer found because case-sensitive compare now correct

5.1.3. UPPER

Figure 5.3. Example UPPER Function

```
select UPPER(c.firstName) from Customer c where c.firstName=LOWER('CAT')
```

```
  select
    upper(customer0_.FIRST_NAME) as col_0_0_
  from JPAQL_CUSTOMER customer0_
  where customer0_.FIRST_NAME=lower('CAT')

-found result:CAT
```

- First name of customer located returned in upper case

5.1.4. TRIM

Figure 5.4. Example TRIM Function

```
select TRIM(LEADING 'c' FROM c.firstName) from Customer c where c.firstName='cat')
```

```
  select
     trim(LEADING 'c' FROM customer0_.FIRST_NAME) as col_0_0_
  from JPAQL_CUSTOMER customer0_
  where customer0_.FIRST_NAME='cat'

-found result:at
```

- Customer's name, excluding initial 'c' character, returned

5.1.5. CONCAT

Figure 5.5. Example CONCAT Function]

```
select c from Customer c where CONCAT(CONCAT(c.firstName,' '),c.lastName) ='cat inhat')
```

```
  select
     customer0_.CUSTOMER_ID as CUSTOMER1_3_,
     customer0_.FIRST_NAME as FIRST2_3_,
     customer0_.LAST_NAME as LAST3_3_
  from JPAQL_CUSTOMER customer0_
  where ( (customer0_.FIRST_NAME||' ')||customer0_.LAST_NAME)='cat inhat'

-found result:firstName=cat, lastName=inhat
```

- Customer located after concatenation of fields yields match

5.1.6. LENGTH

Figure 5.6. Example LENGTH Function

```
select c from Customer c where LENGTH(c.firstName) = 3
```

```
  select
     customer0_.CUSTOMER_ID as CUSTOMER1_3_,
     customer0_.FIRST_NAME as FIRST2_3_,
     customer0_.LAST_NAME as LAST3_3_
  from JPAQL_CUSTOMER customer0_
```

```
    where length(customer0_.FIRST_NAME)=3

 -found result:firstName=cat, lastName=inhat
```

- Customer found where length of firstName matches specified length criteria

5.1.7. LOCATE

Figure 5.7. Example LOCATE Function

```
select c from Customer c where LOCATE('cat',c.firstName,2) > 0
```

```
    select
        customer0_.CUSTOMER_ID as CUSTOMER1_3_,
        customer0_.FIRST_NAME as FIRST2_3_,
        customer0_.LAST_NAME as LAST3_3_
    from JPAQL_CUSTOMER customer0_
    where locate('cat', customer0_.FIRST_NAME, 2)>0
```

- No firstName found with 'cat' starting at position=2

Figure 5.8. Another Example LOCATE Function

```
select c from Customer c where LOCATE('at',c.firstName,2) > 1
```

```
    select
        customer0_.CUSTOMER_ID as CUSTOMER1_3_,
        customer0_.FIRST_NAME as FIRST2_3_,
        customer0_.LAST_NAME as LAST3_3_
    from JPAQL_CUSTOMER customer0_
    where locate('at', customer0_.FIRST_NAME, 2)>1

 -found result:firstName=cat, lastName=inhat
```

- firstName found with 'at' starting at a position 2

5.1.8. SUBSTRING

Figure 5.9. Example SUBSTRING Function

```
select SUBSTRING(c.firstName,2,2) from Customer c where c.firstName = 'cat'
```

```
    select
        substring(customer0_.FIRST_NAME, 2, 2) as col_0_0_
    from JPAQL_CUSTOMER customer0_
    where customer0_.FIRST_NAME='cat'
    -found result:at
```

- Return the two character substring of firstName starting at position two

Figure 5.10. Another Example SUBSTRING Function

```
select c from Customer c where SUBSTRING(c.firstName,2,2) = 'at'
```

```
    select
        customer0_.CUSTOMER_ID as CUSTOMER1_3_,
        customer0_.FIRST_NAME as FIRST2_3_,
        customer0_.LAST_NAME as LAST3_3_
    from JPAQL_CUSTOMER customer0_
    where substring(customer0_.FIRST_NAME, 2, 2)='at'

 -found result:firstName=cat, lastName=inhat
```

- Find the customer with a two characters starting a position two of firstName equaling 'at'

5.2. Date Functions

- CURRENT_DATE
- CURRENT_TIME
- CURRENT_TIMESTAMP

Figure 5.11. CURRENT_DATE Query Example

```
select s from Sale s
where s.date < CURRENT_DATE
```

```
    select
        sale0_.SALE_ID as SALE1_1_,
        sale0_.amount as amount1_,
        sale0_.BUYER_ID as BUYER3_1_,
        sale0_.date as date1_,
        sale0_.SALE_STORE as SALE5_1_
    from JPAQL_SALE sale0_
    where sale0_.date<CURRENT_DATE
...
 -found result:date=1998-04-10 10:13:35, amount=$100.00, buyer=1, clerks(1)={1, }
...
 -found result:date=1999-06-11 14:15:10, amount=$150.00, buyer=2, clerks(2)={1, 2, }
```

- Located two Sales that occurred prior to today's date

Figure 5.12. Another CURRENT_DATE Query Example

```
select s from Sale s
where s.date = CURRENT_DATE
```

```
    select
        sale0_.SALE_ID as SALE1_1_,
        sale0_.amount as amount1_,
        sale0_.BUYER_ID as BUYER3_1_,
        sale0_.date as date1_,
        sale0_.SALE_STORE as SALE5_1_
    from JPAQL_SALE sale0_
    where sale0_.date=CURRENT_DATE
```

- Located no sales on today's date

Figure 5.13. Using Bulk Update to Change Date

```
update Sale s
set s.date = CURRENT_DATE
```

```
update JPAQL_SALE
set date=CURRENT_DATE
```

- Update all sales to today

Figure 5.14. Retrying CURRENT_DATE Query After Bulk Update

```
select s from Sale s
where s.date = CURRENT_DATE
```

```
 -executing query:select s from Sale s where s.date = CURRENT_DATE
    select
        sale0_.SALE_ID as SALE1_1_,
        sale0_.amount as amount1_,
        sale0_.BUYER_ID as BUYER3_1_,
        sale0_.date as date1_,
        sale0_.SALE_STORE as SALE5_1_
    from JPAQL_SALE sale0_
    where sale0_.date=CURRENT_DATE
...
 -found result:date=2013-06-05 00:00:00, amount=$100.00, buyer=1, clerks(1)={1, }
...
 -found result:date=2013-06-05 00:00:00, amount=$150.00, buyer=2, clerks(2)={1, 2, }
```

- Now locating sales for today's date

Note

Bulk commands (i.e., update) invalidate cached entities. You must refresh their state with the database or detach/clear them from the persistence context to avoid using out-dated information.

5.3. Order By

- ASC - ascending order
- DESC - descending order

Figure 5.15. Example Order By

```
select s from Sale s ORDER BY s.amount ASC
```

```
    select
        sale0_.SALE_ID as SALE1_1_,
        sale0_.amount as amount1_,
        sale0_.BUYER_ID as BUYER3_1_,
        sale0_.date as date1_,
        sale0_.SALE_STORE as SALE5_1_
    from JPAQL_SALE sale0_
    order by sale0_.amount ASC
...
 -found result:date=1998-04-10 10:13:35, amount=$100.00, buyer=1, clerks(1)={1, }
...
 -found result:date=1999-06-11 14:15:10, amount=$150.00, buyer=2, clerks(2)={1, 2, }
```

- Note the ASC order on amount

Figure 5.16. Another Example Order By

```
select s from Sale s ORDER BY s.amount DESC
```

```
    select
        sale0_.SALE_ID as SALE1_1_,
        sale0_.amount as amount1_,
        sale0_.BUYER_ID as BUYER3_1_,
        sale0_.date as date1_,
        sale0_.SALE_STORE as SALE5_1_
    from JPAQL_SALE sale0_
    order by sale0_.amount DESC
 -found result:date=1999-06-11 14:15:10, amount=$150.00, buyer=2, clerks(2)={1, 2, }
 -found result:date=1998-04-10 10:13:35, amount=$100.00, buyer=1, clerks(1)={1, }
```

- Note the DESC order on amount

5.4. Aggregate Functions

5.4.1. COUNT

Figure 5.17. Example COUNT Aggregate Function

```
select COUNT(s) from Sale s
```

```
    select count(*) as col_0_0_
    from JPAQL_SALE sale0_

 -found result:2
```

5.4.2. MIN/MAX

Figure 5.18. Example MIN Aggregate Function

```
select min(s.amount) from Sale s
```

```
    select min(sale0_.amount) as col_0_0_
    from JPAQL_SALE sale0_
 -found result:100.00
```

Figure 5.19. Example MAX Aggregate Function

```
select max(s.amount) from Sale s
```

```
    select max(sale0_.amount) as col_0_0_
    from JPAQL_SALE sale0_
 -found result:150.00
```

5.4.3. SUM/AVE

Figure 5.20. Example SUM Aggregate Function

```
select sum(s.amount) from Sale s
```

```
    select sum(sale0_.amount) as col_0_0_
    from JPAQL_SALE sale0_
 -found result:250.00
```

Figure 5.21. Example AVE Aggregate Function

```
select ave(s.amount) from Sale s
```

```
    select avg(cast(sale0_.amount as double)) as col_0_0_
    from JPAQL_SALE sale0_
 -found result:125.0
```

5.5. Group By

Figure 5.22. Example Group By

```
select c, COUNT(s) from Clerk c
LEFT JOIN c.sales s
GROUP BY c
```

- Get count of sales for each clerk

Figure 5.23. Example Group By Runtime Output

```
    select
        clerk0_.CLERK_ID as col_0_0_,
        count(sale2_.SALE_ID) as col_1_0_,
        clerk0_.CLERK_ID as CLERK1_2_,
        clerk0_.FIRST_NAME as FIRST2_2_,
        clerk0_.HIRE_DATE as HIRE3_2_,
        clerk0_.LAST_NAME as LAST4_2_,
        clerk0_.TERM_DATE as TERM5_2_
    from JPAQL_CLERK clerk0_
    left outer join JPAQL_SALE_CLERK_LINK sales1_  on clerk0_.CLERK_ID=sales1_.CLERK_ID
    left outer join JPAQL_SALE sale2_  on sales1_.SALE_ID=sale2_.SALE_ID
    group by clerk0_.CLERK_ID
...
 -found=[firstName=Manny, lastName=Pep, hireDate=1970-01-01, termDate=null, sales(2)={1, 2, }, 2]
...
 -found=[firstName=Moe, lastName=Pep, hireDate=1970-01-01, termDate=null, sales(1)={2, }, 1]
...
 -found=[firstName=Jack, lastName=Pep, hireDate=1973-03-01, termDate=null, sales(0)={}, 0]
```

5.6. Having

Figure 5.24. Example Having Aggregate Function

```
select c, COUNT(s) from Clerk c
LEFT JOIN c.sales s
GROUP BY c
HAVING COUNT(S) <= 1
```

- Provide a list of Clerks and their count of Sales for counts <= 1

Figure 5.25. Example Having Aggregate Function Runtime Output

```

    select
        clerk0_.CLERK_ID as col_0_0_,
        count(sale2_.SALE_ID) as col_1_0_,
        clerk0_.CLERK_ID as CLERK1_2_,
        clerk0_.FIRST_NAME as FIRST2_2_,
        clerk0_.HIRE_DATE as HIRE3_2_,
        clerk0_.LAST_NAME as LAST4_2_,
        clerk0_.TERM_DATE as TERM5_2_
    from JPAQL_CLERK clerk0_
    left outer join JPAQL_SALE_CLERK_LINK sales1_  on clerk0_.CLERK_ID=sales1_.CLERK_ID
    left outer join JPAQL_SALE sale2_ on sales1_.SALE_ID=sale2_.SALE_ID
    group by clerk0_.CLERK_ID
    having count(sale2_.SALE_ID)<=1
...
 -found=[firstName=Moe, lastName=Pep, hireDate=1970-01-01, termDate=null, sales(1)={2, }, 1]
...
```

```
-found=[firstName=Jack, lastName=Pep, hireDate=1973-03-01, termDate=null, sales(0)={}, 0]
```

- Wed matched on Moe (1 sale) and Jack (0 sales)

5.7. Summary

- String and Date Functions
- Order By
- Group By, Having, and Aggregate Functions

Part III. Criteria API

JPA Criteria API

6.1. Criteria API Demo Template

- Following sections contain but may not show the following code in all cases

Figure 6.1. Boiler-plate code

```
CriteriaBuilder cb = em.getCriteriaBuilder();

//example-specific criteria API definition goes here
CriteriaQuery<T> qdef = ...

//repeated display loop eliminated
TypedQuery<T> query = em.createQuery(qdef);
List<T> objects = query.getResultList();
for(T o: objects) {
  log.info("found result:" + o);
}
```

- Get CriteriaBuilder from EntityManager
- Build example-specific query definition in CriteriaQuery using CriteriaBuilder
- Execute Query/Print results

6.2. Simple Entity Query

Figure 6.2. Equivalent JPAQL

```
select c from Customer c
```

Figure 6.3. Criteria API Definition

```
CriteriaQuery<Customer> qdef = cb.createQuery(Customer.class);

Root<Customer> c = qdef.from(Customer.class);
qdef.select(c);
```

- "from"
  - defines source of root query terms
  - returns object leveraged in query body
- "select" defines root query objects -- all path references must start from this set
- no "where" clause indicates all entities are selected

Figure 6.4. In Programming Context

```
CriteriaQuery<Customer> qdef = cb.createQuery(Customer.class);
```

```
Root<Customer> c = qdef.from(Customer.class);
qdef.select(c);

TypedQuery<Customer> query = em.createQuery(qdef);
List<Customer> results = query.getResultList();
```

Figure 6.5. Runtime Output

```
   select
       customer0_.CUSTOMER_ID as CUSTOMER1_3_,
       customer0_.FIRST_NAME as FIRST2_3_,
       customer0_.LAST_NAME as LAST3_3_
   from JPAQL_CUSTOMER customer0_
 -found result:firstName=cat, lastName=inhat
 -found result:firstName=thing, lastName=one
 -found result:firstName=thing, lastName=two
```

6.3. Non-Entity Query

Figure 6.6. Equivalent JPAQL

```
select c.lastName from Customer c
```

Figure 6.7. Criteria API Definition

```
CriteriaQuery<String> qdef = cb.createQuery(String.class);

Root<Customer> c = qdef.from(Customer.class);
qdef.select(c.<String>get("lastName"));
```

- Allows return of simple property
- c.get("lastName") is called a "path"
- All paths based from root query terms (thus requirement for Root<Customer> c object)
- Single path selects return typed list of values

Figure 6.8. In Programming Context

```
CriteriaQuery<String> qdef = cb.createQuery(String.class);

Root<Customer> c = qdef.from(Customer.class);
qdef.select(c.<String>get("lastName"));

TypedQuery<String> query = em.createQuery(qdef);
List<String> results = query.getResultList();
```

- Query result is a List<String> because "c.lastName" is a String

Figure 6.9. Runtime Output

```
  select customer0_.LAST_NAME as col_0_0_
  from JPAQL_CUSTOMER customer0_
-lastName=inhat
-lastName=one
-lastName=two
```

6.4. Multi-select Query

6.4.1. Multi-select Query with Object[]

Figure 6.10. Equivalent JPAQL

```
select c.firstName, c.hireDate from Clerk c
```

Figure 6.11. Criteria API Definition

```
CriteriaQuery<Object[]> qdef = cb.createQuery(Object[].class);

Root<Clerk> c = qdef.from(Clerk.class);
qdef.select(cb.array(c.get("firstName"), c.get("hireDate")));
```

- Select specifies multiple terms within array()
- Terms are expressed thru a path expression
- Terms must be based off paths from root terms in the FROM (or JOIN) clause -- thus why Root<Clerk> c was retained from cb.from() call

Figure 6.12. In Programming Context

```
CriteriaQuery<Object[]> qdef = cb.createQuery(Object[].class);

Root<Clerk> c = qdef.from(Clerk.class);
qdef.select(cb.array(c.get("firstName"), c.get("hireDate")));

TypedQuery<Object[]> query = em.createQuery(qdef);
List<Object[]> results = query.getResultList();
assertTrue("no results", results.size() > 0);
for(Object[] result : results) {
  assertEquals("unexpected result length", 2, result.length);
  String firstName = (String) result[0];
  Date hireDate = (Date) result[1];
  log.info("firstName=" + firstName + " hireDate=" + hireDate);
}
```

- Query defined to return elements of select in Object[]

Figure 6.13. Runtime Output

```
    select
        clerk0_.FIRST_NAME as col_0_0_,
        clerk0_.HIRE_DATE as col_1_0_
    from JPAQL_CLERK clerk0_
-firstName=Manny hireDate=1970-01-01
-firstName=Moe hireDate=1970-01-01
-firstName=Jack hireDate=1973-03-01
```

6.4.2. Multi-select Query with Tuple

Figure 6.14. Equivalent JPAQL

```
select c.firstName as firstName, c.hireDate as hireDate from Clerk c
```

Figure 6.15. Criteria API Definition

```
CriteriaQuery<Tuple> qdef = cb.createTupleQuery();

Root<Clerk> c = qdef.from(Clerk.class);
qdef.select(cb.tuple(
        c.get("firstName").alias("firstName"),
        c.get("hireDate").alias("hireDate")));
```

- Aliases may be assigned to select terms for named-access to results

Figure 6.16. In Programming Context

```
CriteriaQuery<Tuple> qdef = cb.createTupleQuery();

Root<Clerk> c = qdef.from(Clerk.class);
qdef.select(cb.tuple(
        c.get("firstName").alias("firstName"),
        c.get("hireDate").alias("hireDate")));

TypedQuery<Tuple> query = em.createQuery(qdef);
List<Tuple> results = query.getResultList();
assertTrue("no results", results.size() > 0);
for(Tuple result : results) {
    assertEquals("unexpected result length", 2, result.getElements().size());
    String firstName = result.get("firstName", String.class);
    Date hireDate = result.get("hireDate", Date.class);
    log.info("firstName=" + firstName + " hireDate=" + hireDate);
}
```

- Query defined to return instances of Tuple class
- Tuples provide access using
  - get(index) - simular to Object[]
  - get(index, Class<T> resultType) - typed access by index

- get(alias) - access by alias
- get(alias, Class<T> resultType) - typed access by alias
- getElements() - access thru collection interface

Figure 6.17. Runtime Output

```
  select
    clerk0_.FIRST_NAME as col_0_0_,
    clerk0_.HIRE_DATE as col_1_0_
  from JPAQL_CLERK clerk0_
-firstName=Manny hireDate=1970-01-01
-firstName=Moe hireDate=1970-01-01
-firstName=Jack hireDate=1973-03-01
```

6.4.3. Multi-select Query with Constructor

Figure 6.18. Equivalent JPAQL

```
select new ejava.jpa.examples.query.Receipt(s.id,s.buyerId,s.date, s.amount)
from Sale s
```

Figure 6.19. Criteria API Definition

```
CriteriaQuery<Receipt> qdef = cb.createQuery(Receipt.class);

Root<Sale> s = qdef.from(Sale.class);
qdef.select(cb.construct(
        Receipt.class,
        s.get("id"),
        s.get("buyerId"),
        s.get("date"),
        s.get("amount")));
```

- Individual elements of select() are matched up against class constructor

Figure 6.20. In Programming Context

```
CriteriaQuery<Receipt> qdef = cb.createQuery(Receipt.class);

Root<Sale> s = qdef.from(Sale.class);
qdef.select(cb.construct(
        Receipt.class,
        s.get("id"),
        s.get("buyerId"),
        s.get("date"),
        s.get("amount")));

TypedQuery<Receipt> query = em.createQuery(qdef);
List<Receipt> results = query.getResultList();
assertTrue("no results", results.size() > 0);
```

```
for(Receipt receipt : results) {
  assertNotNull("no receipt", receipt);
  log.info("receipt=" + receipt);
}
```

- Constructed class may be simple POJO -- no need to be an entity
- Instances are not managed
- Suitable for use as Data Transfer Objects (DTOs)

Figure 6.21. Runtime Output

```
  select
    sale0_.SALE_ID as col_0_0_,
    sale0_.BUYER_ID as col_1_0_,
    sale0_.date as col_2_0_,
    sale0_.amount as col_3_0_
  from JPAQL_SALE sale0_
-receipt=sale=1, customer=1, date=1998-04-10 10:13:35, amount=$100.00
-receipt=sale=2, customer=2, date=1999-06-11 14:15:10, amount=$150.00
```

6.5. Path Expressions

6.5.1. Single Element Path Expressions

Figure 6.22. Equivalent JPAQL

```
select s.id, s.store.name from Sale s
```

Figure 6.23. Criteria API Definition

```
CriteriaQuery<Object[]> qdef = cb.createQuery(Object[].class);

Root<Sale> s = qdef.from(Sale.class);
qdef.select(cb.array(s.get("id"),
           s.get("store").get("name")));
```

- All paths based off root-level FROM (or JOIN) terms
- Paths use call chaining to change contexts
- Paths -- used this way -- must always express a single element. Must use JOINs for paths involving collections
- All paths based off root-level FROM (or JOIN) terms

Figure 6.24. In Programming Context

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Object[]> qdef = cb.createQuery(Object[].class);

Root<Sale> s = qdef.from(Sale.class);
```

```
qdef.select(cb.array(s.get("id"),
                     s.get("store").get("name")));

TypedQuery<Object[]> query = em.createQuery(qdef);
List<Object[]> results = query.getResultList();
assertTrue("no results", results.size() > 0);
for(Object[] result : results) {
    assertEquals("unexpected result length", 2, result.length);
    Long id = (Long) result[0];
    String name = (String) result[1];
    log.info("sale.id=" + id + ", sale.store.name=" + name);
}
```

Figure 6.25. Runtime Output

```
    select
        sale0_.SALE_ID as col_0_0_,
        store1_.name as col_1_0_
    from JPAQL_SALE sale0_, ORMQL_STORE store1_
    where sale0_.SALE_STORE=store1_.STORE_ID
 -sale.id=1, sale.store.name=Big Al's
 -sale.id=2, sale.store.name=Big Al's
```

- Automatic INNER JOIN formed between Sale and Store because of the cross-entity path

6.5.2. Collection Element Path Expressions

6.5.2.1. INNER JOIN Collection Path Expressions

Figure 6.26. Equivalent JPAQL

```
select sale.date from Clerk c JOIN c.sales sale
```

Figure 6.27. Criteria API Definition

```
CriteriaQuery<Date> qdef = cb.createQuery(Date.class);

Root<Clerk> c = qdef.from(Clerk.class);
Join<Clerk, Sale> sale = c.join("sales", JoinType.INNER);
qdef.select(sale.<Date>get("date"));
```

- Collection is brought in as a root term of the query through a JOIN expression
- JOINs will match entities by their defined primary/foreign keys
- INNER JOIN will return only those entities where there is a match
- INNER JOIN is default JoinType when none specified

Figure 6.28. Runtime Output

```
    select sale2_.date as col_0_0_
    from JPAQL_CLERK clerk0_
    inner join JPAQL_SALE_CLERK_LINK sales1_ on clerk0_.CLERK_ID=sales1_.CLERK_ID
    inner join JPAQL_SALE sale2_ on sales1_.SALE_ID=sale2_.SALE_ID

 -found result:1998-04-10 10:13:35.0
 -found result:1999-06-11 14:15:10.0
 -found result:1999-06-11 14:15:10.0
```

- (Many-to-Many) Link table used during JOIN
- Tables automatically joined on primary keys
- Only Sales sold by our Clerks are returned

6.5.2.2. LEFT OUTER JOIN Collection Path Expressions

Figure 6.29. Equivalent JPAQL

```
select c.id, c.firstName, sale.amount
from Clerk c
LEFT JOIN c.sales sale
```

Figure 6.30. Criteria API Definition

```
CriteriaQuery<Object[]> qdef = cb.createQuery(Object[].class);

Root<Clerk> c = qdef.from(Clerk.class);
Join<Clerk, Sale> sale = c.join("sales", JoinType.LEFT);
qdef.select(cb.array(c.get("id"),
                     c.get("firstName"),
                     sale.get("amount")));
```

- LEFT OUTER JOIN will return root with or without related entities

Figure 6.31. Runtime Output

```
    select
        clerk0_.CLERK_ID as col_0_0_,
        clerk0_.FIRST_NAME as col_1_0_,
        sale2_.amount as col_2_0_
    from JPAQL_CLERK clerk0_
    left outer join JPAQL_SALE_CLERK_LINK sales1_ on clerk0_.CLERK_ID=sales1_.CLERK_ID
    left outer join JPAQL_SALE sale2_ on sales1_.SALE_ID=sale2_.SALE_ID
 -clerk.id=1, clerk.firstName=Manny, amount=100.00
 -clerk.id=1, clerk.firstName=Manny, amount=150.00
 -clerk.id=2, clerk.firstName=Moe, amount=150.00
 -clerk.id=3, clerk.firstName=Jack, amount=null
```

- (Many-to-Many) Link table used during JOIN
- Tables automatically joined on primary keys
- All clerks, with or without a Sale, are returned

6.5.2.3. Explicit Collection Path Expressions

Figure 6.32. Equivalent JPAQL

```
select c from Sale s, Customer c
where c.id = s.buyerId
```

Figure 6.33. Criteria API Definition

```
CriteriaQuery<Customer> qdef = cb.createQuery(Customer.class);

Root<Sale> s = qdef.from(Sale.class);
Root<Customer> c = qdef.from(Customer.class);
qdef.select(c)
    .where(cb.equal(c.get("id"), s.get("buyerId")));
```

- Permits JOINs without relationship in entity model

Figure 6.34. Runtime Output

```
    select
        customer1_.CUSTOMER_ID as CUSTOMER1_3_,
        customer1_.FIRST_NAME as FIRST2_3_,
        customer1_.LAST_NAME as LAST3_3_
    from JPAQL_SALE sale0_ cross
    join JPAQL_CUSTOMER customer1_
    where customer1_.CUSTOMER_ID=sale0_.BUYER_ID

 -found result:firstName=cat, lastName=inhat
 -found result:firstName=thing, lastName=one
```

- Returns all Customers that are identified by a Sale

6.6. Eager Fetching through JOINs

6.6.1. Lazy Fetch Problem

Figure 6.35. Equivalent JPAQL

```
select s from Store s JOIN s.sales
where s.name='Big Al''s'
```

Figure 6.36. Criteria API Definition

```
CriteriaQuery<Store> qdef = cb.createQuery(Store.class);

Root<Store> s = qdef.from(Store.class);
s.join("sales");
qdef.select(s)
```

```
    .where(cb.equal(s.get("name"), "Big Al's"));
```

- A normal JOIN (implicit or explicit) may honor the fetch=LAZY property setting of the relation
- Can be exactly what is desired
- Can also cause problems or extra work if not desired

Figure 6.37. In Programming Context

```
@Entity @Table(name="ORMQL_STORE")
public class Store {
...
    @OneToMany(mappedBy="store",
            cascade={CascadeType.REMOVE},
            fetch=FetchType.LAZY)
    private List<Sale> sales = new ArrayList<Sale>();
```

- Sales are lazily fetched when obtaining Store

Figure 6.38. In Programming Context (con.t)

```
CriteriaBuilder cb = em2.getCriteriaBuilder();
CriteriaQuery<Store> qdef = cb.createQuery(Store.class);

Root<Store> s = qdef.from(Store.class);
s.join("sales");
qdef.select(s)
    .where(cb.equal(s.get("name"), "Big Al's"));

Store store = em2.createQuery(qdef).getSingleResult();
em2.close();
try {
    store.getSales().get(0).getAmount();
    fail("did not trigger lazy initialization exception");
} catch (LazyInitializationException expected) {
    log.info("caught expected exception:" + expected);
}
```

- Accessing the Sale properties causes a LazyInitializationException when persistence context no longer active or accessible

Figure 6.39. Runtime Output

```
    select
        store0_.STORE_ID as STORE1_0_,
        store0_.name as name0_
    from ORMQL_STORE store0_
    inner join JPAQL_SALE sales1_ on store0_.STORE_ID=sales1_.SALE_STORE
    where store0_.name=? limit ?
 -caught expected exception:org.hibernate.LazyInitializationException:
     failed to lazily initialize a collection of role:
     ejava.jpa.examples.query.Store.sales, no session or session was closed
```

One Row per Parent is Returned for fetch=LAZY

Note that only a single row is required to be returned from the database for a fetch=LAZY relation. Although it requires more queries to the database, it eliminates duplicate parent information for each child row and can eliminate the follow-on query all together when not accessed.

6.6.2. Adding Eager Fetch during Query

Figure 6.40. Equivalent JPAQL

```
select s from Store s JOIN FETCH s.sales
where s.name='Big Al''s'
```

Figure 6.41. Criteria API Definition

```
CriteriaQuery<Store> qdef = cb.createQuery(Store.class);

Root<Store> s = qdef.from(Store.class);
s.fetch("sales");
qdef.select(s)
    .where(cb.equal(s.get("name"), "Big Al's"));
```

- A JOIN FETCH used to eager load related entities as side-effect of query
- Can be used as substitute for fetch=EAGER specification on relation

Figure 6.42. Runtime Output

```
    select
        store0_.STORE_ID as STORE1_0_0_,
        sales1_.SALE_ID as SALE1_1_1_,
        store0_.name as name0_0_,
        sales1_.amount as amount1_1_,
        sales1_.BUYER_ID as BUYER3_1_1_,
        sales1_.date as date1_1_,
        sales1_.SALE_STORE as SALE5_1_1_,
        sales1_.SALE_STORE as SALE5_0_0__,
        sales1_.SALE_ID as SALE1_0__
    from ORMQL_STORE store0_
    inner join JPAQL_SALE sales1_ on store0_.STORE_ID=sales1_.SALE_STORE
    where store0_.name=?
```

- Sales are eagerly fetched when obtaining Store

Parent Rows Repeated for each Child for fetch=EAGER

Note that adding JOIN FETCH to parent query causes the parent rows to be repeated for each eagerly loaded child row and eliminated by the provider. This requires fewer database queries but results in more (and redundant) data to be returned from the query.

6.7. Distinct Results

Figure 6.43. Equivalent JPAQL

```
select DISTINCT c.lastName from Customer c
```

Figure 6.44. Criteria API Definition

```
CriteriaQuery<String> qdef = cb.createQuery(String.class);

Root<Customer> c = qdef.from(Customer.class);
qdef.select(c.<String>get("lastName"))
  .distinct(true);
```

- Limits output to unique value combinations

Figure 6.45. Runtime Output

```
  select distinct customer0_.LAST_NAME as col_0_0_
  from JPAQL_CUSTOMER customer0_
 -found result:two
 -found result:inhat
 -found result:one
```

Figure 6.46. Equivalent JPAQL

```
select DISTINCT c.firstName from Customer c
```

Figure 6.47. Criteria API Definition

```
CriteriaQuery<String> qdef = cb.createQuery(String.class);

Root<Customer> c = qdef.from(Customer.class);
qdef.select(c.<String>get("firstName"))
  .distinct(true);
```

Figure 6.48. Runtime Output

```
  select distinct customer0_.FIRST_NAME as col_0_0_
  from JPAQL_CUSTOMER customer0_
 -found result:cat
 -found result:thing
```

6.8. Summary

- Entity Queries
- Non-Entity (value) and Multi-select Queries
- Path Expressions (Root and Join)
- Join Fetch
- Distinct Results

Criteria Where Clauses

7.1. Equality Test

Figure 7.1. Equivalent JPAQL

```
select c from Customer c
where c.firstName='cat'
```

Figure 7.2. Criteria API Definition

```
CriteriaQuery<Customer> qdef = cb.createQuery(Customer.class);

Root<Customer> c = qdef.from(Customer.class);
qdef.select(c)
    .where(cb.equal(c.get("firstName"), "cat"));
```

- Return entities where there is an equality match

Figure 7.3. Runtime Output

```
    select
        customer0_.CUSTOMER_ID as CUSTOMER1_3_,
        customer0_.FIRST_NAME as FIRST2_3_,
        customer0_.LAST_NAME as LAST3_3_
    from JPAQL_CUSTOMER customer0_
    where customer0_.FIRST_NAME=?
 -found result:firstName=cat, lastName=inhat
```

Figure 7.4. Equivalent JPAQL

```
select s from Store s
where s.name='Big Al''s'
```

- JPAQL requires special characters to be escaped

Figure 7.5. Criteria API Definition

```
CriteriaQuery<Store> qdef = cb.createQuery(Store.class);

Root<Store> s = qdef.from(Store.class);
qdef.select(s)
    .where(cb.equal(s.get("name"), "Big Al's"));
```

- Literal values automatically escaped

Figure 7.6. Runtime Output

```
    select
        store0_.STORE_ID as STORE1_0_,
        store0_.name as name0_
    from ORMQL_STORE store0_
    where store0_.name=?
...
 -found result:name=Big Al's, sales(2)={1, 2, }
```

7.2. Like Test

7.2.1. Like Test Literal

Figure 7.7. Equivalent JPAQL

```
select c from Clerk c
where c.firstName like 'M%'
```

Figure 7.8. Criteria API Definition

```
CriteriaQuery<Clerk> qdef = cb.createQuery(Clerk.class);

Root<Clerk> c = qdef.from(Clerk.class);
qdef.select(c)
    .where(cb.like(c.<String>get("firstName"), "M%"));
```

Figure 7.9. Runtime Output

```
    select
        clerk0_.CLERK_ID as CLERK1_2_,
        clerk0_.FIRST_NAME as FIRST2_2_,
        clerk0_.HIRE_DATE as HIRE3_2_,
        clerk0_.LAST_NAME as LAST4_2_,
        clerk0_.TERM_DATE as TERM5_2_
    from JPAQL_CLERK clerk0_
    where clerk0_.FIRST_NAME like ?
...
 -found result:firstName=Manny, lastName=Pep, hireDate=1970-01-01, termDate=null, sales(2)={1, 2, }
 -found result:firstName=Moe, lastName=Pep, hireDate=1970-01-01, termDate=null, sales(1)={2, }
```

7.2.2. Like Test Literal Parameter

Figure 7.10. Equivalent JPAQL

```
select c from Clerk c
where c.firstName like :firstName
```

Figure 7.11. Criteria API Definition

```
CriteriaQuery<Clerk> qdef = cb.createQuery(Clerk.class);

Root<Clerk> c = qdef.from(Clerk.class);
qdef.select(c)
    .where(cb.like(c.<String>get("firstName"),
                   cb.parameter(String.class, "firstName")));
```

Figure 7.12. In Programming Context

```
CriteriaQuery<Clerk> qdef = cb.createQuery(Clerk.class);

Root<Clerk> c = qdef.from(Clerk.class);
qdef.select(c)
    .where(cb.like(c.<String>get("firstName"),
                   cb.parameter(String.class, "firstName")));

TypedQuery<Clerk> query = em.createQuery(qdef)
        .setParameter("firstName", "M%");
List<Clerk> results = query.getResultList();
```

Figure 7.13. Runtime Output

```
    select
        clerk0_.CLERK_ID as CLERK1_2_,
        clerk0_.FIRST_NAME as FIRST2_2_,
        clerk0_.HIRE_DATE as HIRE3_2_,
        clerk0_.LAST_NAME as LAST4_2_,
        clerk0_.TERM_DATE as TERM5_2_
    from JPAQL_CLERK clerk0_
    where clerk0_.FIRST_NAME like ?
 -found result:firstName=Manny, lastName=Pep, hireDate=1970-01-01, termDate=null, sales(2)={1, 2, }
 -found result:firstName=Moe, lastName=Pep, hireDate=1970-01-01, termDate=null, sales(1)={2, }
```

7.2.3. Like Test Concatenated String

Figure 7.14. Equivalent JPAQL

```
select c from Clerk c
where c.firstName like concat(:firstName,'%')
```

Figure 7.15. Criteria API Definition

```
CriteriaQuery<Clerk> qdef = cb.createQuery(Clerk.class);

Root<Clerk> c = qdef.from(Clerk.class);
qdef.select(c)
    .where(cb.like(c.<String>get("firstName"),
                   cb.concat(cb.parameter(String.class, "firstName"), "%")));
```

Figure 7.16. In Programming Context

```
CriteriaQuery<Clerk> qdef = cb.createQuery(Clerk.class);

Root<Clerk> c = qdef.from(Clerk.class);
qdef.select(c)
    .where(cb.like(c.<String>get("firstName"),
                   cb.concat(cb.parameter(String.class, "firstName"), "%")));

TypedQuery<Clerk> query = em.createQuery(qdef)
        .setParameter("firstName", "M");
List<Clerk> results = query.getResultList();
```

Figure 7.17. Runtime Output

```
    select
        clerk0_.CLERK_ID as CLERK1_2_,
        clerk0_.FIRST_NAME as FIRST2_2_,
        clerk0_.HIRE_DATE as HIRE3_2_,
        clerk0_.LAST_NAME as LAST4_2_,
        clerk0_.TERM_DATE as TERM5_2_
    from JPAQL_CLERK clerk0_
    where clerk0_.FIRST_NAME like (?||?)
 -found result:firstName=Manny, lastName=Pep, hireDate=1970-01-01, termDate=null, sales(2)={1, 2, }
 -found result:firstName=Moe, lastName=Pep, hireDate=1970-01-01, termDate=null, sales(1)={2, }
```

7.2.4. Like Test Single Character Wildcard

Figure 7.18. Equivalent JPAQL

```
select c from Clerk c
where c.firstName like '_anny'
```

Figure 7.19. Criteria API Definition

```
CriteriaQuery<Clerk> qdef = cb.createQuery(Clerk.class);

Root<Clerk> c = qdef.from(Clerk.class);
qdef.select(c)
    .where(cb.like(c.<String>get("firstName"),"_anny"));
```

Figure 7.20. In Programming Context

```
CriteriaQuery<Clerk> qdef = cb.createQuery(Clerk.class);

Root<Clerk> c = qdef.from(Clerk.class);
qdef.select(c)
    .where(cb.like(c.<String>get("firstName"),"_anny"));

TypedQuery<Clerk> query = em.createQuery(qdef);
```

```
List<Clerk> results = query.getResultList();
```

Figure 7.21. Runtime Output

```
    select
        clerk0_.CLERK_ID as CLERK1_2_,
        clerk0_.FIRST_NAME as FIRST2_2_,
        clerk0_.HIRE_DATE as HIRE3_2_,
        clerk0_.LAST_NAME as LAST4_2_,
        clerk0_.TERM_DATE as TERM5_2_
    from JPAQL_CLERK clerk0_
    where clerk0_.FIRST_NAME like ?

 -found result:firstName=Manny, lastName=Pep, hireDate=1970-01-01, termDate=null, sales(2)={1, 2, }
```

7.3. Formulas

Figure 7.22. Equivalent JPAQL

```
select s from Sale s
where (s.amount * :tax) > :amount
```

Figure 7.23. Criteria API Definition

```
CriteriaQuery<Number> qdef = cb.createQuery(Number.class);

Root<Sale> s = qdef.from(Sale.class);
qdef.select(cb.count(s))
    .where(cb.greaterThan(
        cb.prod(s.<BigDecimal>get("amount"), cb.parameter(BigDecimal.class, "tax")),
        new BigDecimal(10.0)));
```

Figure 7.24. In Programming Context

```
CriteriaQuery<Number> qdef = cb.createQuery(Number.class);

//select count(s) from Sale s
//where (s.amount * :tax) > :amount"
Root<Sale> s = qdef.from(Sale.class);
qdef.select(cb.count(s))
    .where(cb.greaterThan(
        cb.prod(s.<BigDecimal>get("amount"), cb.parameter(BigDecimal.class, "tax")),
        new BigDecimal(10.0)));
TypedQuery<Number> query = em.createQuery(qdef);

//keep raising taxes until somebody pays $10.00 in tax
double tax = 0.05;
for (;query.setParameter("tax", new BigDecimal(tax))
          .getSingleResult().intValue()==0;
      tax += 0.01) {
    log.debug("tax=" + NumberFormat.getPercentInstance().format(tax));
}
log.info("raise taxes to: " + NumberFormat.getPercentInstance().format(tax));
```

Figure 7.25. Runtime Output

```

  select count(*) as col_0_0_
  from JPAQL_SALE sale0_
  where sale0_.amount*?>10 limit ?
-tax=5%
  select count(*) as col_0_0_
  from JPAQL_SALE sale0_
  where sale0_.amount*?>10 limit ?
-tax=6%
  select count(*) as col_0_0_
  from JPAQL_SALE sale0_
  where sale0_.amount*?>10 limit ?
-raise taxes to: 7%
```

7.4. Logic Operators

Figure 7.26. Equivalent JPAQL

```
select c from Customer c
where (c.firstName='cat' AND c.lastName='inhat')
  OR c.firstName='thing'
```

Figure 7.27. Criteria API Definition

```
CriteriaQuery<Customer> qdef = cb.createQuery(Customer.class);

Root<Customer> c = qdef.from(Customer.class);
qdef.select(c)
  .where(cb.or(
      cb.and(cb.equal(c.get("firstName"), "cat"),
         cb.equal(c.get("lastName"), "inhat")),
      cb.equal(c.get("firstName"), "thing")));
```

Figure 7.28. Runtime Output

```
  select
    customer0_.CUSTOMER_ID as CUSTOMER1_3_,
    customer0_.FIRST_NAME as FIRST2_3_,
    customer0_.LAST_NAME as LAST3_3_
  from JPAQL_CUSTOMER customer0_
  where customer0_.FIRST_NAME=? and customer0_.LAST_NAME=? or customer0_.FIRST_NAME=?
-found result:firstName=cat, lastName=inhat
-found result:firstName=thing, lastName=one
-found result:firstName=thing, lastName=two
```

Figure 7.29. Equivalent JPAQL

```
select c from Customer c
where (NOT (c.firstName='cat' AND c.lastName='inhat'))
   OR c.firstName='thing'
```

Figure 7.30. Criteria API Definition

```
CriteriaQuery<Customer> qdef = cb.createQuery(Customer.class);

Root<Customer> c = qdef.from(Customer.class);
qdef.select(c)
   .where(cb.or(
         cb.not(cb.and(cb.equal(c.get("firstName"), "cat"),
                    cb.equal(c.get("lastName"), "inhat"))),
         cb.equal(c.get("firstName"), "thing"))
      );
```

Figure 7.31. Runtime Output

```
   select
      customer0_.CUSTOMER_ID as CUSTOMER1_3_,
      customer0_.FIRST_NAME as FIRST2_3_,
      customer0_.LAST_NAME as LAST3_3_
   from JPAQL_CUSTOMER customer0_
   where customer0_.FIRST_NAME<>? or customer0_.LAST_NAME<>? or customer0_.FIRST_NAME=?
 -found result:firstName=thing, lastName=one
 -found result:firstName=thing, lastName=two
```

7.5. Equality Tests

- Must compare values
  - Of same type
  - Of legal promotion type
    - Can compare 123:int to 123:long
    - Cannot compare 123:int to "123":string
  - Can compare entities

Figure 7.32. Equivalent JPAQL

```
select c from Clerk c
where c.firstName = 'Manny'
```

```
select s from Sale s JOIN s.clerks c
where c = :clerk
```

Figure 7.33. Criteria API Definition

```
CriteriaQuery<Clerk> qdef = cb.createQuery(Clerk.class);
```

```
Root<Clerk> c = qdef.from(Clerk.class);
qdef.select(c)
   .where(cb.equal(c.get("firstName"), "Manny"));
```

```

CriteriaQuery<Sale> qdef2 = cb.createQuery(Sale.class);
Root<Sale> s = qdef2.from(Sale.class);
Join<Sale, Clerk> c2 = s.join("clerks");
qdef2.select(s)
    .where(cb.equal(c2, clerk));
```

- Compare entities and not primary/foreign key values

Figure 7.34. In Programming Context

```
//find clerk of interest
CriteriaQuery<Clerk> qdef = cb.createQuery(Clerk.class);
Root<Clerk> c = qdef.from(Clerk.class);
qdef.select(c)
   .where(cb.equal(c.get("firstName"), "Manny"));
Clerk clerk = em.createQuery(qdef).getSingleResult();

//find all sales that involve this clerk
CriteriaQuery<Sale> qdef2 = cb.createQuery(Sale.class);
Root<Sale> s = qdef2.from(Sale.class);
Join<Sale, Clerk> c2 = s.join("clerks");
qdef2.select(s)
    .where(cb.equal(c2, clerk));

List<Sale> sales = em.createQuery(qdef2).getResultList();
```

Figure 7.35. Runtime Output

```
  select
    clerk0_.CLERK_ID as CLERK1_2_,
    clerk0_.FIRST_NAME as FIRST2_2_,
    clerk0_.HIRE_DATE as HIRE3_2_,
    clerk0_.LAST_NAME as LAST4_2_,
    clerk0_.TERM_DATE as TERM5_2_
  from JPAQL_CLERK clerk0_
  where clerk0_.FIRST_NAME='Manny' limit ?
```

```
  select
    sale0_.SALE_ID as SALE1_1_,
    sale0_.amount as amount1_,
    sale0_.BUYER_ID as BUYER3_1_,
    sale0_.date as date1_,
    sale0_.SALE_STORE as SALE5_1_
  from JPAQL_SALE sale0_
  inner join JPAQL_SALE_CLERK_LINK clerks1_ on sale0_.SALE_ID=clerks1_.SALE_ID
  inner join JPAQL_CLERK clerk2_ on clerks1_.CLERK_ID=clerk2_.CLERK_ID
  where clerk2_.CLERK_ID=?
...
```

```
 -found=date=1998-04-10 10:13:35, amount=$100.00, buyer=1, clerks(1)={1, }
...
 -found=date=1999-06-11 14:15:10, amount=$150.00, buyer=2, clerks(2)={1, 2, }
```

7.6. Between

Figure 7.36. Equivalent JPAQL

```
select s from Sale s
where s.amount BETWEEN :low AND :high
```

Figure 7.37. Criteria API Definition

```
CriteriaQuery<Sale> qdef = cb.createQuery(Sale.class);
Root<Sale> s = qdef.from(Sale.class);
qdef.select(s)
    .where(cb.between(s.<BigDecimal>get("amount"),
            new BigDecimal(90.00),
            new BigDecimal(110.00)));
```

Figure 7.38. Runtime Output

```
    select
        sale0_.SALE_ID as SALE1_1_,
        sale0_.amount as amount1_,
        sale0_.BUYER_ID as BUYER3_1_,
        sale0_.date as date1_,
        sale0_.SALE_STORE as SALE5_1_
    from JPAQL_SALE sale0_
    where sale0_.amount between 90 and 110
...
 -found=date=1998-04-10 10:13:35, amount=$100.00, buyer=1, clerks(1)={1, }
```

Figure 7.39. Equivalent JPAQL

```
select s from Sale s
where s.amount NOT BETWEEN :low AND :high
```

Figure 7.40. Criteria API Definition

```
CriteriaQuery<Sale> qdef = cb.createQuery(Sale.class);
Root<Sale> s = qdef.from(Sale.class);
qdef.select(s)
    .where(cb.not(cb.between(s.<BigDecimal>get("amount"),
            new BigDecimal(90.00),
            new BigDecimal(110.00))));
```

Figure 7.41. Runtime Output

```
  select
    sale0_.SALE_ID as SALE1_1_,
    sale0_.amount as amount1_,
    sale0_.BUYER_ID as BUYER3_1_,
    sale0_.date as date1_,
    sale0_.SALE_STORE as SALE5_1_
  from JPAQL_SALE sale0_
  where sale0_.amount not between 90 and 110
...
 -found=date=1999-06-11 14:15:10, amount=$150.00, buyer=2, clerks(2)={1, 2, }
```

7.7. Testing for Null

Can be used to test for unassigned value or relationship

Figure 7.42. Equivalent JPAQL

```
select s from Sale s
where s.store IS NULL
```

Figure 7.43. Criteria API Definition

```
CriteriaQuery<Sale> qdef = cb.createQuery(Sale.class);
Root<Sale> s = qdef.from(Sale.class);
qdef.select(s)
  .where(cb.isNull(s.get("store")));
  //.where(cb.equal(s.get("store"), cb.nullLiteral(Store.class)));
```

Figure 7.44. Runtime Output

```
  select
    sale0_.SALE_ID as SALE1_1_,
    sale0_.amount as amount1_,
    sale0_.BUYER_ID as BUYER3_1_,
    sale0_.date as date1_,
    sale0_.SALE_STORE as SALE5_1_
  from JPAQL_SALE sale0_
  where sale0_.SALE_STORE is null
```

Figure 7.45. Equivalent JPAQL

```
select s from Sale s
where s.store IS NOT NULL
```

Figure 7.46. Criteria API Definition

```
CriteriaQuery<Sale> qdef = cb.createQuery(Sale.class);
```

```
Root<Sale> s = qdef.from(Sale.class);
qdef.select(s)
  .where(cb.isNotNull(s.get("store")));
  //.where(cb.not(cb.equal(s.get("store"), cb.nullLiteral(Store.class))));
```

Figure 7.47. Runtime Output

```
  select
    sale0_.SALE_ID as SALE1_1_,
    sale0_.amount as amount1_,
    sale0_.BUYER_ID as BUYER3_1_,
    sale0_.date as date1_,
    sale0_.SALE_STORE as SALE5_1_
  from JPAQL_SALE sale0_
  where sale0_.SALE_STORE is not null
...
 -found result:date=1998-04-10 10:13:35, amount=$100.00, buyer=1, clerks(1)={1, }
...
 -found result:date=1999-06-11 14:15:10, amount=$150.00, buyer=2, clerks(2)={1, 2, }
```

7.8. Testing Empty Collection

Can be used to test for an empty collection

Figure 7.48. Equivalent JPAQL

```
select c from Clerk c
where c.sales IS EMPTY
```

Figure 7.49. Criteria API Definition

```
CriteriaQuery<Clerk> qdef = cb.createQuery(Clerk.class);
Root<Clerk> c = qdef.from(Clerk.class);
qdef.select(c)
  .where(cb.isEmpty(c.<List<Sale>>get("sales")));
```

Figure 7.50. Runtime Output

```
  select
    clerk0_.CLERK_ID as CLERK1_2_,
    clerk0_.FIRST_NAME as FIRST2_2_,
    clerk0_.HIRE_DATE as HIRE3_2_,
    clerk0_.LAST_NAME as LAST4_2_,
    clerk0_.TERM_DATE as TERM5_2_
  from JPAQL_CLERK clerk0_
  where not (exists (
    select sale2_.SALE_ID
    from JPAQL_SALE_CLERK_LINK sales1_, JPAQL_SALE sale2_
    where clerk0_.CLERK_ID=sales1_.CLERK_ID and sales1_.SALE_ID=sale2_.SALE_ID)
  )
...
```

```
 -found result:firstName=Jack, lastName=Pep, hireDate=1973-03-01, termDate=null, sales(0)={}
```

- Sub-select returns values from collection under test
- Outer query tests for no existing (EMPTY)values

Figure 7.51. Equivalent JPAQL

```
select c from Clerk c
where c.sales IS NOT EMPTY
```

Figure 7.52. Criteria API Definition

```
CriteriaQuery<Clerk> qdef = cb.createQuery(Clerk.class);

Root<Clerk> c = qdef.from(Clerk.class);
qdef.select(c)
    .where(cb.isNotEmpty(c.<List<Sale>>get("sales")));
```

Figure 7.53. Runtime Output

```
    select
        clerk0_.CLERK_ID as CLERK1_2_,
        clerk0_.FIRST_NAME as FIRST2_2_,
        clerk0_.HIRE_DATE as HIRE3_2_,
        clerk0_.LAST_NAME as LAST4_2_,
        clerk0_.TERM_DATE as TERM5_2_
    from JPAQL_CLERK clerk0_
    where exists (
            select sale2_.SALE_ID
            from JPAQL_SALE_CLERK_LINK sales1_, JPAQL_SALE sale2_
            where clerk0_.CLERK_ID=sales1_.CLERK_ID and sales1_.SALE_ID=sale2_.SALE_ID
        )
...
 -found result:firstName=Manny, lastName=Pep, hireDate=1970-01-01, termDate=null, sales(2)={1, 2, }
...
 -found result:firstName=Moe, lastName=Pep, hireDate=1970-01-01, termDate=null, sales(1)={2, }
```

- Sub-select returns values from collection under test
- Outer query tests for existing (NOT EMPTY)values

7.9. Membership Test

Can be used to determine membership in a collection

Figure 7.54. Equivalent JPAQL

```
select c from Clerk c
where c.firstName = 'Manny'
```

```
select s from Sale s
```

```
where :clerk MEMBER OF s.clerks
```

Figure 7.55. Criteria API Definition

```
CriteriaQuery<Clerk> qdef = cb.createQuery(Clerk.class);
Root<Clerk> c = qdef.from(Clerk.class);
qdef.select(c)
   .where(cb.equal(c.get("firstName"), "Manny"));
```

```
CriteriaQuery<Sale> qdef2 = cb.createQuery(Sale.class);
Root<Sale> s = qdef2.from(Sale.class);
qdef2.select(s)
    .where(cb.isMember(clerk, s.<List<Clerk>>get("clerks")));
```

- Defines a shorthand for a subquery

Figure 7.56. In Programming Context

```
CriteriaQuery<Clerk> qdef = cb.createQuery(Clerk.class);
Root<Clerk> c = qdef.from(Clerk.class);
qdef.select(c)
   .where(cb.equal(c.get("firstName"), "Manny"));
Clerk clerk = em.createQuery(qdef).getSingleResult();

//find all sales that involve this clerk
CriteriaQuery<Sale> qdef2 = cb.createQuery(Sale.class);
Root<Sale> s = qdef2.from(Sale.class);
qdef2.select(s)
    .where(cb.isMember(clerk, s.<List<Clerk>>get("clerks")));
List<Sale> sales = em.createQuery(qdef2).getResultList();
```

Figure 7.57. Runtime Output

```
   select
      clerk0_.CLERK_ID as CLERK1_2_,
      clerk0_.FIRST_NAME as FIRST2_2_,
      clerk0_.HIRE_DATE as HIRE3_2_,
      clerk0_.LAST_NAME as LAST4_2_,
      clerk0_.TERM_DATE as TERM5_2_
   from JPAQL_CLERK clerk0_
   where clerk0_.FIRST_NAME=? limit ?
```

```
   select
      sale0_.SALE_ID as SALE1_1_,
      sale0_.amount as amount1_,
      sale0_.BUYER_ID as BUYER3_1_,
      sale0_.date as date1_,
      sale0_.SALE_STORE as SALE5_1_
   from JPAQL_SALE sale0_
   where ? in (
         select clerk2_.CLERK_ID
         from JPAQL_SALE_CLERK_LINK clerks1_, JPAQL_CLERK clerk2_
      where sale0_.SALE_ID=clerks1_.SALE_ID and clerks1_.CLERK_ID=clerk2_.CLERK_ID
    )
...
 -found=date=1998-04-10 10:13:35, amount=$100.00, buyer=1, clerks(1)={1, }
...
 -found=date=1999-06-11 14:15:10, amount=$150.00, buyer=2, clerks(2)={1, 2, }
```

7.10. Subqueries

Useful when query cannot be expressed through JOINs

Figure 7.58. Equivalent JPAQL

```
select c from Customer c
where c.id IN
  (select s.buyerId from Sale s
   where s.amount > 100)
```

Figure 7.59. Criteria API Definition

```
CriteriaQuery<Customer> qdef = cb.createQuery(Customer.class);

  //form subquery
Subquery<Long> sqdef = qdef.subquery(Long.class);
Root<Sale> s = sqdef.from(Sale.class);
sqdef.select(s.<Long>get("buyerId"))
  .where(cb.greaterThan(s.<BigDecimal>get("amount"), new BigDecimal(100)));

  //form outer query
Root<Customer> c = qdef.from(Customer.class);
qdef.select(c)
  .where(cb.in(c.get("id")).value(sqdef));
```

Figure 7.60. Runtime Output

```
  select
    customer0_.CUSTOMER_ID as CUSTOMER1_3_,
    customer0_.FIRST_NAME as FIRST2_3_,
    customer0_.LAST_NAME as LAST3_3_
  from JPAQL_CUSTOMER customer0_
  where customer0_.CUSTOMER_ID in (
      select sale1_.BUYER_ID
      from JPAQL_SALE sale1_
      where sale1_.amount>100
    )

 -found result:firstName=thing, lastName=one
```

7.11. All

- All existing values must meet criteria (i.e., no value may fail criteria)

- Zero values is the lack of failure (i.e., meets criteria)

Figure 7.61. Equivalent JPAQL

```
select c from Clerk c
where 125 < ALL
(select s.amount from c.sales s)
```

Figure 7.62. Criteria API Definition

```
CriteriaQuery<Clerk> qdef = cb.createQuery(Clerk.class);
Root<Clerk> c = qdef.from(Clerk.class);
qdef.select(c);

Subquery<BigDecimal> sqdef = qdef.subquery(BigDecimal.class);
Root<Clerk> c1 = sqdef.from(Clerk.class);
Join<Clerk,Sale> s = c1.join("sales");
sqdef.select(s.<BigDecimal>get("amount"))
    .where(cb.equal(c, c1));

Predicate p1 = cb.lessThan(
        cb.literal(new BigDecimal(125)),
        cb.all(sqdef));
qdef.where(p1);
```

- List all clerks that have all sales above $125.00 or none at all
- -or- List all clerks with no sale <= $125.00

Figure 7.63. Runtime Output

```
    select
        clerk0_.CLERK_ID as CLERK1_2_,
        clerk0_.FIRST_NAME as FIRST2_2_,
        clerk0_.HIRE_DATE as HIRE3_2_,
        clerk0_.LAST_NAME as LAST4_2_,
        clerk0_.TERM_DATE as TERM5_2_
    from JPAQL_CLERK clerk0_
    where 125<all (
            select sale3_.amount
            from JPAQL_CLERK clerk1_
            inner join JPAQL_SALE_CLERK_LINK sales2_ on clerk1_.CLERK_ID=sales2_.CLERK_ID
            inner join JPAQL_SALE sale3_ on sales2_.SALE_ID=sale3_.SALE_ID
            where clerk0_.CLERK_ID=clerk1_.CLERK_ID
        )
...
 -found result:firstName=Moe, lastName=Pep, hireDate=1970-01-01, termDate=null, sales(1)={2, }
...
 -found result:firstName=Jack, lastName=Pep, hireDate=1973-03-01, termDate=null, sales(0)={}
```

- Manny excluded because has 1 sale below $125.00
- Moe included because has only $150.00 sale

- Jack included because has no sales that fail criteria

Figure 7.64. Equivalent JPAQL

```
select c from Clerk c
where 125 > ALL
(select s.amount from c.sales s)
```

Figure 7.65. Criteria API Definition

```
Predicate p2 = cb.greaterThan(
        cb.literal(new BigDecimal(125)),
        cb.all(sqdef));

qdef.where(p2);
```

- List all clerks that have all sales below $125.00 or none at all
- -or- List all clerks with no sale >= $125.00

Figure 7.66. Runtime Output

```
    select
        clerk0_.CLERK_ID as CLERK1_2_,
        clerk0_.FIRST_NAME as FIRST2_2_,
        clerk0_.HIRE_DATE as HIRE3_2_,
        clerk0_.LAST_NAME as LAST4_2_,
        clerk0_.TERM_DATE as TERM5_2_
    from JPAQL_CLERK clerk0_
    where 125>all (
            select sale3_.amount
            from JPAQL_CLERK clerk1_
            inner join JPAQL_SALE_CLERK_LINK sales2_ on clerk1_.CLERK_ID=sales2_.CLERK_ID
            inner join JPAQL_SALE sale3_ on sales2_.SALE_ID=sale3_.SALE_ID
            where clerk0_.CLERK_ID=clerk1_.CLERK_ID
        )

 -found result:firstName=Jack, lastName=Pep, hireDate=1973-03-01, termDate=null, sales(0)={}
```

- Manny excluded because has 1 sale above $125.00
- Moe excluded because has only $150.00 sale
- Jack included because has no sales that fail criteria

7.12. Any

- Any matching value meets criteria (i.e., one match and you are in)
- Zero values fails to meet the criteria (i.e., must have at least one matching value)

Figure 7.67. Equivalent JPAQL

```
select c from Clerk c
```

```
where 125 < ANY
(select s.amount from c.sales s)
```

Figure 7.68. Criteria API Definition

```
CriteriaQuery<Clerk> qdef = cb.createQuery(Clerk.class);
Root<Clerk> c = qdef.from(Clerk.class);
qdef.select(c);

//select c from Clerk c
//where 125 < ALL " +
//(select s.amount from c.sales s)",
Subquery<BigDecimal> sqdef = qdef.subquery(BigDecimal.class);
Root<Clerk> c1 = sqdef.from(Clerk.class);
Join<Clerk,Sale> s = c1.join("sales");
sqdef.select(s.<BigDecimal>get("amount"))
   .where(cb.equal(c, c1));

Predicate p1 = cb.lessThan(
    cb.literal(new BigDecimal(125)),
    cb.any(sqdef));

qdef.where(p1);
```

- List all clerks that have at least one sale above $125.00

Figure 7.69. Runtime Output

```
  select
    clerk0_.CLERK_ID as CLERK1_2_,
    clerk0_.FIRST_NAME as FIRST2_2_,
    clerk0_.HIRE_DATE as HIRE3_2_,
    clerk0_.LAST_NAME as LAST4_2_,
    clerk0_.TERM_DATE as TERM5_2_
  from JPAQL_CLERK clerk0_
  where 125<any (
      select sale3_.amount
      from JPAQL_CLERK clerk1_
      inner join JPAQL_SALE_CLERK_LINK sales2_ on clerk1_.CLERK_ID=sales2_.CLERK_ID
      inner join JPAQL_SALE sale3_ on sales2_.SALE_ID=sale3_.SALE_ID
      where clerk0_.CLERK_ID=clerk1_.CLERK_ID
    )
...
 -found result:firstName=Manny, lastName=Pep, hireDate=1970-01-01, termDate=null, sales(2)={1, 2, }
...
 -found result:firstName=Moe, lastName=Pep, hireDate=1970-01-01, termDate=null, sales(1)={2, }
```

- Manny included because has 1 sale above $125.00
- Moe included because $150.00 sale qualifies him as well
- Jack excluded because has no sales that meet criteria

Figure 7.70. Equivalent JPAQL

```
select c from Clerk c
where 125 > ANY
(select s.amount from c.sales s)
```

Figure 7.71. Criteria API Definition

```
Predicate p2 = cb.greaterThan(
    cb.literal(new BigDecimal(125)),
    cb.any(sqdef));

qdef.where(p2);
```

- List all clerks that have at least one sale below $125.00

Figure 7.72. Runtime Output

```
  select
    clerk0_.CLERK_ID as CLERK1_2_,
    clerk0_.FIRST_NAME as FIRST2_2_,
    clerk0_.HIRE_DATE as HIRE3_2_,
    clerk0_.LAST_NAME as LAST4_2_,
    clerk0_.TERM_DATE as TERM5_2_
  from JPAQL_CLERK clerk0_
  where 125>any (
      select sale3_.amount
      from JPAQL_CLERK clerk1_
      inner join JPAQL_SALE_CLERK_LINK sales2_ on clerk1_.CLERK_ID=sales2_.CLERK_ID
      inner join JPAQL_SALE sale3_ on sales2_.SALE_ID=sale3_.SALE_ID
      where clerk0_.CLERK_ID=clerk1_.CLERK_ID
    )

-found result:firstName=Manny, lastName=Pep, hireDate=1970-01-01, termDate=null, sales(2)={1, 2, }
```

- Manny included because has 1 sale below $125.00
- Moe excluded because his only $150.00 sale above criteria
- Jack excluded because has no sales that meet criteria

7.13. Summary

- Full featured set of where clause capability

Chapter 8.

Criteria Functions

8.1. String Functions

8.1.1. Base Query

Figure 8.1. Equivalent JPAQL

```
select c from Customer c where c.firstName='CAT'
```

Figure 8.2. Criteria API Definition

```
CriteriaQuery qdef = cb.createQuery();
Root<Customer> c = qdef.from(Customer.class);

qdef.select(c)
    .where(cb.equal(c.get("firstName"),"CAT"));
```

- Using an untyped CriteriaQuery to be able to switch between different query output types within example

Figure 8.3. Runtime Output

```
    select
        customer0_.CUSTOMER_ID as CUSTOMER1_3_,
        customer0_.FIRST_NAME as FIRST2_3_,
        customer0_.LAST_NAME as LAST3_3_
    from JPAQL_CUSTOMER customer0_
    where customer0_.FIRST_NAME=?
```

- No rows found because 'CAT' does not match anything because of case

8.1.2. LOWER

Figure 8.4. Equivalent JPAQL

```
select c from Customer c where c.firstName=LOWER('CAT')
```

Figure 8.5. Criteria API Definition

```
qdef.select(c)
    .where(cb.equal(c.get("firstName"),cb.lower(cb.literal("CAT"))));
```

Figure 8.6. Runtime Output

```
    select
```

```
        customer0_.CUSTOMER_ID as CUSTOMER1_3_,
        customer0_.FIRST_NAME as FIRST2_3_,
        customer0_.LAST_NAME as LAST3_3_
    from JPAQL_CUSTOMER customer0_
    where customer0_.FIRST_NAME=lower(?)

 -found result:firstName=cat, lastName=inhat
```

- One customer found because case-sensitive compare now correct

8.1.3. UPPER

Figure 8.7. Equivalent JPAQL

```
select UPPER(c.firstName) from Customer c where c.firstName=LOWER('CAT')
```

Figure 8.8. Criteria API Definition

```
qdef.select(cb.upper(c.<String>get("firstName")))
    .where(cb.equal(c.get("firstName"),cb.lower(cb.literal("CAT"))));
```

Figure 8.9. Runtime Output

```
    select upper(customer0_.FIRST_NAME) as col_0_0_
    from JPAQL_CUSTOMER customer0_
    where customer0_.FIRST_NAME=lower(?)

 -found result:CAT
```

- First name of customer located returned in upper case

8.1.4. TRIM

Figure 8.10. Equivalent JPAQL

```
select TRIM(LEADING 'c' FROM c.firstName) from Customer c where c.firstName='cat')
```

Figure 8.11. Criteria API Definition

```
qdef.select(cb.trim(Trimspec.LEADING, 'c', c.<String>get("firstName")))
    .where(cb.equal(c.get("firstName"),"cat"));
```

Figure 8.12. Runtime Output

```
    select trim(LEADING ?
    from customer0_.FIRST_NAME) as col_0_0_
    from JPAQL_CUSTOMER customer0_
    where customer0_.FIRST_NAME=?
```

```
 -found result:at
```

- Customer's name, excluding initial 'c' character, returned

8.1.5. CONCAT

Figure 8.13. Equivalent JPAQL

```
select c from Customer c where CONCAT(CONCAT(c.firstName,' '),c.lastName) ='cat inhat')
```

Figure 8.14. Criteria API Definition

```
qdef.select(c)
    .where(cb.equal(
            cb.concat(
                    cb.concat(c.<String>get("firstName"), " "),
                    c.<String>get("lastName")),
            "cat inhat"));
```

Figure 8.15. Runtime Output

```
    select
        customer0_.CUSTOMER_ID as CUSTOMER1_3_,
        customer0_.FIRST_NAME as FIRST2_3_,
        customer0_.LAST_NAME as LAST3_3_
    from JPAQL_CUSTOMER customer0_
    where (customer0_.FIRST_NAME||?||customer0_.LAST_NAME)=?

 -found result:firstName=cat, lastName=inhat
```

- Customer located after concatenation of fields yields match

8.1.6. LENGTH

Figure 8.16. Equivalent JPAQL

```
select c from Customer c where LENGTH(c.firstName) = 3
```

Figure 8.17. Criteria API Definition

```
qdef.select(c)
    .where(cb.equal(cb.length(c.<String>get("firstName")),3));
```

Figure 8.18. Runtime Output

```
    select
        customer0_.CUSTOMER_ID as CUSTOMER1_3_,
        customer0_.FIRST_NAME as FIRST2_3_,
        customer0_.LAST_NAME as LAST3_3_ 
    from JPAQL_CUSTOMER customer0_ 
    where length(customer0_.FIRST_NAME)=3

 -found result:firstName=cat, lastName=inhat
```

- Customer found where length of firstName matches specified length criteria

8.1.7. LOCATE

Figure 8.19. Equivalent JPAQL

```
select c from Customer c where LOCATE('cat',c.firstName,2) > 0
```

Figure 8.20. Criteria API Definition

```
qdef.select(c)
    .where(cb.greaterThan(cb.locate(c.<String>get("firstName"), "cat", 2),0));
```

Figure 8.21. Runtime Output

```
    select
        customer0_.CUSTOMER_ID as CUSTOMER1_3_,
        customer0_.FIRST_NAME as FIRST2_3_,
        customer0_.LAST_NAME as LAST3_3_ 
    from JPAQL_CUSTOMER customer0_ 
    where locate(?, customer0_.FIRST_NAME, 2)>0
```

- No firstName found with 'cat' starting at position=2

Figure 8.22. Equivalent JPAQL

```
select c from Customer c where LOCATE('at',c.firstName,2) > 1
```

```
qdef.select(c)
    .where(cb.greaterThan(cb.locate(c.<String>get("firstName"), "at", 2),1));
```

Figure 8.23. Runtime Output

```
    select
        customer0_.CUSTOMER_ID as CUSTOMER1_3_,
        customer0_.FIRST_NAME as FIRST2_3_,
        customer0_.LAST_NAME as LAST3_3_ 
    from JPAQL_CUSTOMER customer0_ 
    where locate(?, customer0_.FIRST_NAME, 2)>1

 -found result:firstName=cat, lastName=inhat
```

- firstName found with 'at' starting at a position 2

8.1.8. SUBSTRING

Figure 8.24. Equivalent JPAQL

```
select SUBSTRING(c.firstName,2,2) from Customer c where c.firstName = 'cat'
```

Figure 8.25. Criteria API Definition

```
qdef.select(cb.substring(c.<String>get("firstName"),  2, 2))
    .where(cb.equal(c.get("firstName"), "cat"));
```

Figure 8.26. Runtime Output

```
    select substring(customer0_.FIRST_NAME, 2, 2) as col_0_0_ 
    from JPAQL_CUSTOMER customer0_ 
    where customer0_.FIRST_NAME=?

 -found result:at
```

- Return the two character substring of firstName starting at position two

Figure 8.27. Equivalent JPAQL

```
select c from Customer c where SUBSTRING(c.firstName,2,2) = 'at'
```

Figure 8.28. Criteria API Definition

```
qdef.select(c)
    .where(cb.equal(
            cb.substring(c.<String>get("firstName"), 2, 2), 
            "at"));
```

Figure 8.29. Runtime Output

```
    select
        customer0_.CUSTOMER_ID as CUSTOMER1_3_,
        customer0_.FIRST_NAME as FIRST2_3_,
        customer0_.LAST_NAME as LAST3_3_ 
    from JPAQL_CUSTOMER customer0_ 
    where substring(customer0_.FIRST_NAME, 2, 2)=?
 -found result:firstName=cat, lastName=inhat
```

- Find the customer with a two characters starting a position two of firstName equaling 'at'

8.2. Date Functions

- CriteriaBuilder.currentDate()
- CriteriaBuilder.currentTime()

- CriteriaBuilder.currentTimestamp()

Figure 8.30. Equivalent JPAQL

```
select s from Sale s
where s.date < CURRENT_DATE
```

Figure 8.31. Criteria API Definition

```
CriteriaQuery<Sale> qdef = cb.createQuery(Sale.class);
Root<Sale> s = qdef.from(Sale.class);
qdef.select(s);

qdef.where(cb.lessThan(s.<Date>get("date"), cb.currentDate()));
```

Figure 8.32. Runtime Output

```
  select
    sale0_.SALE_ID as SALE1_1_,
    sale0_.amount as amount1_,
    sale0_.BUYER_ID as BUYER3_1_,
    sale0_.date as date1_,
    sale0_.SALE_STORE as SALE5_1_
  from JPAQL_SALE sale0_
  where sale0_.date<current_date()
...
 -found result:date=1998-04-10 10:13:35, amount=$100.00, buyer=1, clerks(1)={1, }
...
 -found result:date=1999-06-11 14:15:10, amount=$150.00, buyer=2, clerks(2)={1, 2, }
```

- Located two Sales that occurred prior to today's date

Figure 8.33. Equivalent JPAQL

```
select s from Sale s
where s.date = CURRENT_DATE
```

Figure 8.34. Criteria API Definition

```
qdef.where(cb.equal(s.<Date>get("date"), cb.currentDate()));
```

Figure 8.35. Runtime Output

```
  select
    sale0_.SALE_ID as SALE1_1_,
    sale0_.amount as amount1_,
    sale0_.BUYER_ID as BUYER3_1_,
    sale0_.date as date1_,
    sale0_.SALE_STORE as SALE5_1_
  from JPAQL_SALE sale0_
```

```
    where sale0_.date=current_date()
```

- Located no sales on today's date
- Update with a bulk query

> **Note**
>
> Criteria API added Bulk Updates in JPA 2.1

Figure 8.36. Equivalent JPAQL

```
update Sale s set s.date = CURRENT_DATE
```

Figure 8.37. Criteria API Definition

```
CriteriaUpdate<Sale> qupdate = cb.createCriteriaUpdate(Sale.class);
Root<Sale> s2 = qupdate.from(Sale.class);
qupdate.set(s2.<Date>get("date"), cb.currentDate());
int rows = em.createQuery(qupdate).executeUpdate();
```

Figure 8.38. Runtime Output

```
update JPAQL_SALE set date=CURRENT_DATE
```

- Update all sales to today

Figure 8.39. Equivalent JPAQL

```
select s from Sale s
where s.date = CURRENT_DATE
```

Figure 8.40. Criteria API Definition

```
qdef.where(cb.equal(s.<Date>get("date"), cb.currentDate()));
```

Figure 8.41. Runtime Output

```
    select
        sale0_.SALE_ID as SALE1_1_,
        sale0_.amount as amount1_,
        sale0_.BUYER_ID as BUYER3_1_,
        sale0_.date as date1_,
        sale0_.SALE_STORE as SALE5_1_
    from JPAQL_SALE sale0_
    where sale0_.date=current_date()
...
 -found result:date=2013-06-05 00:00:00, amount=$100.00, buyer=1, clerks(1)={1, }
...
```

```
 -found result:date=2013-06-05 00:00:00, amount=$150.00, buyer=2, clerks(2)={1, 2, }
```

- Now locating sales for today's date

> **Note**
>
> Bulk commands (i.e., update) invalidate cached entities. You must refresh their state with the database or detach/clear them from the persistence context to avoid using out-dated information.

8.3. Order By

- ASC - ascending order
- DESC - descending order

Figure 8.42. Equivalent JPAQL

```
select s from Sale s ORDER BY s.amount ASC
```

Figure 8.43. Criteria API Definition

```
CriteriaQuery<Sale> qdef = cb.createQuery(Sale.class);
Root<Sale> s = qdef.from(Sale.class);
qdef.select(s);

qdef.orderBy(cb.asc(s.get("amount")));
```

Figure 8.44. Runtime Output

```
    select
        sale0_.SALE_ID as SALE1_1_,
        sale0_.amount as amount1_,
        sale0_.BUYER_ID as BUYER3_1_,
        sale0_.date as date1_,
        sale0_.SALE_STORE as SALE5_1_
    from JPAQL_SALE sale0_
    order by sale0_.amount ASC
...
 -found result:date=1998-04-10 10:13:35, amount=$100.00, buyer=1, clerks(1)={1, }
...
 -found result:date=1999-06-11 14:15:10, amount=$150.00, buyer=2, clerks(2)={1, 2, }
```

- Note the ASC order on amount

Figure 8.45. Equivalent JPAQL

```
select s from Sale s ORDER BY s.amount DESC
```

Figure 8.46. Criteria API Definition

```
qdef.orderBy(cb.desc(s.get("amount")));
```

Figure 8.47. Runtime Output

```
  select
    sale0_.SALE_ID as SALE1_1_,
    sale0_.amount as amount1_,
    sale0_.BUYER_ID as BUYER3_1_,
    sale0_.date as date1_,
    sale0_.SALE_STORE as SALE5_1_
  from JPAQL_SALE sale0_
  order by sale0_.amount DESC
-found result:date=1999-06-11 14:15:10, amount=$150.00, buyer=2, clerks(2)={1, 2, }
-found result:date=1998-04-10 10:13:35, amount=$100.00, buyer=1, clerks(1)={1, }
```

- Note the DESC order on amount

8.4. Aggregate Functions

8.4.1. COUNT

Figure 8.48. Equivalent JPAQL

```
select COUNT(s) from Sale s
```

Figure 8.49. Criteria API Definition

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Number> qdef = cb.createQuery(Number.class);
Root<Sale> s = qdef.from(Sale.class);

qdef.select(cb.count(s));
```

Figure 8.50. Runtime Output

```
select count(*) as col_0_0_
from JPAQL_SALE sale0_
```

8.4.2. MIN/MAX

Figure 8.51. Equivalent JPAQL

```
select min(s.amount) from Sale s
```

Figure 8.52. Criteria API Definition

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Number> qdef = cb.createQuery(Number.class);
Root<Sale> s = qdef.from(Sale.class);

qdef.select(cb.min(s.<BigDecimal>get("amount")));
```

Figure 8.53. Runtime Output

```
    select min(sale0_.amount) as col_0_0_
    from JPAQL_SALE sale0_
 -found result:100.00
```

Figure 8.54. Equivalent JPAQL

```
select max(s.amount) from Sale s
```

Figure 8.55. Criteria API Definition

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Number> qdef = cb.createQuery(Number.class);
Root<Sale> s = qdef.from(Sale.class);

qdef.select(cb.max(s.<BigDecimal>get("amount")));
```

Figure 8.56. Runtime Output

```
    select max(sale0_.amount) as col_0_0_
    from JPAQL_SALE sale0_
 -found result:150.00
```

8.4.3. SUM/AVE

Figure 8.57. Equivalent JPAQL

```
select sum(s.amount) from Sale s
```

Figure 8.58. Criteria API Definition

```
CriteriaQuery<Number> qdef = cb.createQuery(Number.class);
Root<Sale> s = qdef.from(Sale.class);
//select sum(s.amount) from Sale s
qdef.select(cb.sum(s.<BigDecimal>get("amount")));
```

Figure 8.59. Runtime Output

```
  select sum(sale0_.amount) as col_0_0_
  from JPAQL_SALE sale0_
-found result:250.0
```

Figure 8.60. Equivalent JPAQL

```
select ave(s.amount) from Sale s
```

Figure 8.61. Criteria API Definition

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Number> qdef = cb.createQuery(Number.class);
Root<Sale> s = qdef.from(Sale.class);

qdef.select(cb.avg(s.<BigDecimal>get("amount")));
```

Figure 8.62. Runtime Output

```
  select avg(cast(sale0_.amount as double)) as col_0_0_
  from JPAQL_SALE sale0_
-found result:125.0
```

8.5. Group By

Figure 8.63. Equivalent JPAQL

```
select c, COUNT(s) from Clerk c
LEFT JOIN c.sales s
GROUP BY c
```

Figure 8.64. Criteria API Definition

```
CriteriaQuery<Object[]> qdef = cb.createQuery(Object[].class);
Root<Clerk> c = qdef.from(Clerk.class);
Join<Clerk,Sale> s = c.join("sales", JoinType.LEFT);

qdef.select(cb.array(c, cb.count(s)))
  .groupBy(c);
```

- Get count of sales for each clerk

Figure 8.65. Runtime Output

```
  select
    clerk0_.CLERK_ID as col_0_0_,
    count(sale2_.SALE_ID) as col_1_0_,
    clerk0_.CLERK_ID as CLERK1_2_,
    clerk0_.FIRST_NAME as FIRST2_2_,
    clerk0_.HIRE_DATE as HIRE3_2_,
    clerk0_.LAST_NAME as LAST4_2_,
    clerk0_.TERM_DATE as TERM5_2_
  from JPAQL_CLERK clerk0_
  left outer join JPAQL_SALE_CLERK_LINK sales1_ on clerk0_.CLERK_ID=sales1_.CLERK_ID
  left outer join JPAQL_SALE sale2_ on sales1_.SALE_ID=sale2_.SALE_ID
  group by clerk0_.CLERK_ID
...
 -found=[firstName=Manny, lastName=Pep, hireDate=1970-01-01, termDate=null, sales(2)={1, 2, }, 2]
...
 -found=[firstName=Moe, lastName=Pep, hireDate=1970-01-01, termDate=null, sales(1)={2, }, 1]
...
 -found=[firstName=Jack, lastName=Pep, hireDate=1973-03-01, termDate=null, sales(0)={}, 0]
```

8.6. Having

Figure 8.66. Equivalent JPAQL

```
select c, COUNT(s) from Clerk c
LEFT JOIN c.sales s
GROUP BY c
HAVING COUNT(S) <= 1
```

Figure 8.67. Criteria API Definition

```
CriteriaQuery<Object[]> qdef = cb.createQuery(Object[].class);
Root<Clerk> c = qdef.from(Clerk.class);
Join<Clerk,Sale> s = c.join("sales", JoinType.LEFT);

qdef.select(cb.array(c, cb.count(s)))
  .groupBy(c)
  .having(cb.le(cb.count(s), 1));
```

- Provide a list of Clerks and their count of Sales for counts <= 1

Figure 8.68. Runtime Output

```

  select
    clerk0_.CLERK_ID as col_0_0_,
    count(sale2_.SALE_ID) as col_1_0_,
    clerk0_.CLERK_ID as CLERK1_2_,
    clerk0_.FIRST_NAME as FIRST2_2_,
    clerk0_.HIRE_DATE as HIRE3_2_,
    clerk0_.LAST_NAME as LAST4_2_,
    clerk0_.TERM_DATE as TERM5_2_
  from JPAQL_CLERK clerk0_
  left outer join JPAQL_SALE_CLERK_LINK sales1_ on clerk0_.CLERK_ID=sales1_.CLERK_ID
  left outer join JPAQL_SALE sale2_ on sales1_.SALE_ID=sale2_.SALE_ID
  group by clerk0_.CLERK_ID
```

```
    having count(sale2_.SALE_ID)<=1
...
 -found=[firstName=Moe, lastName=Pep, hireDate=1970-01-01, termDate=null, sales(1)={2, }, 1]
...
 -found=[firstName=Jack, lastName=Pep, hireDate=1973-03-01, termDate=null, sales(0)={}, 0]
```

- Wed matched on Moe (1 sale) and Jack (0 sales)

8.7. Summary

- Property and Date Functions
- Order By
- Group By, Aggregate Functions, and Having