JPA Query Guide (v5.1)

Table of Contents

Once you have persisted objects you need to query them. For example if you have a web application representing an online store, the user asks to see all products of a particular type, ordered by the price. This requires you to query the datastore for these products. JPA specifies support for

- JPQL : a string-based query language between SQL and OO.
- Criteria : following JPQL syntax but providing an API supporting refactoring of classes and the queries they are used in.
- Native : equates to SQL when using RDBMS, and CQL when using Cassandra.
- Stored Procedures : in-datastore invocation of stored procedures for RDBMS datastores.

Which query language is used is down to the developer. The data-tier of an application could be written by a primarily Java developer, who would typically think in an object-oriented way and so would likely prefer **JPQL**. On the other hand the data-tier could be written by a datastore developer who is more familiar with SQL concepts and so could easily make more use of **SQL**. This is the power of an implementation like DataNucleus in that it provides the flexibility for different people to develop the data-tier utilising their own skills to the full without having to learn totally new concepts.

There are 2 categories of queries with JPA :-

- **Programmatic Query** where the query is defined using the JPA Query API.
- **Named Query** where the query is defined in MetaData and referred to by its name at runtime(for JPQL, Native Query and Stored Procedures).

Query API

Let's now try to understand the Query API in JPA Javadoc. We firstly need to look at a typical Query. We'll take 2 examples

Let's create a JPQL query to highlight its usage

```
Query q = em.createQuery("SELECT p FROM Product p WHERE p.param2 < :threshold ORDER BY
p.param1 ASC");
q.setParameter("threshold", my_threshold);
List results = q.getResultList();
```

In this Query, we implicitly select JPQL by using the method *EntityManager.createQuery()*, and the query is specified to return all objects of type *Product* (or subclasses) which have the field *param2* less than some threshold value ordering the results by the value of field *param1*. We've specified the query like this because we want to pass the threshold value in as a parameter (so maybe running it once with one value, and once with a different value). We then set the parameter value of our *threshold* parameter. The Query is then executed to return a List of results. The example is to highlight the typical methods specified for a (JPQL) Query.

And for a second example we create a native (SQL) query

```
Query q = em.createNativeQuery("SELECT * FROM Product p WHERE p.param2 < ?1");
q.setParameter(1, my_threshold);
List results = q.getResultList();
```

So we implicitly select SQL by using the method *EntityManager.createNativeQuery()*, and the query is specified like in the JPQL case to return all instances of type *Product* (using the table name in this SQL query) where the column *param2* is less than some threshold value.

setFirstResult(), setMaxResults()

A query will by default return all of the results that it finds. You can restrict how many results are returned by use of two methods. So you could do

```
Query q = em.createQuery("SELECT p FROM Product p WHERE p.param2 < :threshold ORDER BY
p.param1 ASC");
q.setFirstResult(1);
q.setMaxResults(3);
```

so we will get results 1, 2, and 3 returned only. The first result starts at 0 by default.

setHint()

JPA's query API allows implementations to support extensions ("hints") and provides a simple

interface for enabling the use of such extensions on queries.

```
q.setHint("{extension_name}", value);
```

JPA supports some standard hints, namely **javax.persistence.fetchgraph**, **javax.persistence.loadgraph**, **javax.persistence.query.timeout**, **javax.persistence.lock.timeout**. DataNucleus provides various vendor-specific hints for different types of queries (see different parts of this documentation).

setParameter()

When queries take values (literals) it is usually best practice to define these as parameters. JPA's query API supports named and numbered parameters and provides method for setting the value of particular parameters. To set a named parameter, for example, you could do

```
Query q = em.createQuery("SELECT p FROM Product p WHERE p.param2 < :threshold ORDER BY
p.param1 ASC");
q.setParameter("threshold", value);
```

To set a numbered parameter you could do

```
Query q = em.createQuery("SELECT p FROM Product p WHERE p.param2 < ?1 ORDER BY
p.param1 ASC");
q.setParameter(1, value);
```

Numbered parameters are numbered from 1.

getResultList()

To execute a JPA query you would typically call *getResultList*. This will return a List of results. This should not be called when the query is an "UPDATE"/"DELETE".

```
Query q = em.createQuery("SELECT p FROM Product p WHERE p.param2 < :threshold ORDER BY
p.param1 ASC");
q.setParameter("threshold", value);
List results = q.getResultList();
```

getSingleResult()

To execute a JPA query where you are expecting a single value to be returned you would call *getSingleResult*. This will return the single Object. If the query returns more than one result then you will get a *NonUniqueResultException*. This should not be called when the query is an "UPDATE"/"DELETE".

```
Query q = em.createQuery("SELECT p FROM Product p WHERE p.param2 = :value");
q.setParameter("value", val1);
Product prod = q.getSingleResult();
```

executeUpdate()

To execute a JPA UPDATE/DELETE query you would call *executeUpdate*. This will return the number of objects changed by the call. This should not be called when the query is a "SELECT".

```
Query q = em.createQuery("DELETE FROM Product p");
int number = q.executeUpdate();
```

setFlushMode()

By default, when a query is executed it will be evaluated against the contents of the datastore at the point of execution. If there are any outstanding changes waiting to be flushed then these will not feature in the results. To make sure all outstanding changes are respected

```
q.setFlushMode(FlushModeType.AUTO);
```

setLockMode()

JPA allows control over whether objects found by a fetch (JPQL query) are locked during that transaction so that other transactions can't update them in the meantime. For example

```
q.setLockMode(LockModeType.PESSIMISTIC_READ);
```

You can also specify this for all queries for all EntityManagers using a persistence property **datanucleus.rdbms.useUpdateLock**.

Large Result Sets : Loading Results at Commit()

When a transaction is committed by default all remaining results for a query are loaded so that the query is usable thereafter. With a large result set you clearly don't want this to happen. So in this case you should set the query hint **datanucleus.query.loadResultsAtCommit** to *false*, like this

```
query.setHint("datanucleus.query.loadResultsAtCommit", "false");
```

Result Set : Caching of Results

When you execute a query, the query results are typically loaded when the user accesses each row. Results that have been read can then be cached locally. You can control this caching to optimise it for your memory requirements. You can set the query hint **datanucleus.query.resultCacheType** and it has the following possible values

- *weak* : use a weak reference map for caching (default)
- *soft* : use a soft reference map for caching
- *strong* : use a Map for caching (objects not garbage collected)
- *none* : no caching (hence uses least memory)

To do this on a per query basis, you would do

```
query.setHint("datanucleus.query.resultCacheType", "weak");
```

Large Result Sets : Size

If you have a large result set you clearly don't want to instantiate all objects since this would hit the memory footprint of your application. To get the number of results many JDBC drivers, for example, will load all rows of the result set. This is to be avoided so DataNucleus provides control over the mechanism for getting the size of results. The persistence property **datanucleus.query.resultSizeMethod** has a default of *last* (which means navigate to the last object, hence hitting the JDBC driver problem). On RDBMS, if you set this to *count* then it will use a simple "count()" query to get the size.

To do this on a per query basis you would do

```
query.setHint("datanucleus.query.resultSizeMethod", "count");
```

RDBMS : Result Set Type

For RDBMS datastores, *java.sql.ResultSet* defines three possible result set types.

- *forward-only* : the result set is navegable forwards only
- *scroll-sensitive* : the result set is scrollable in both directions and is sensitive to changes in the datastore
- *scroll-insensitive* : the result set is scrollable in both directions and is insensitive to changes in

the datastore

DataNucleus allows specification of this type as a query extension **datanucleus.rdbms.query.resultSetType**.

To do this on a per query basis you would do

```
query.setHint("datanucleus.rdbms.query.resultSetType", "scroll-insensitive");
```

The default is *forward-only*. The benefit of the other two is that the result set will be scrollable and hence objects will only be read in to memory when accessed. So if you have a large result set you should set this to one of the scrollable values.

RDBMS : Result Set Control

DataNucleus RDBMS provides a useful extension allowing control over the ResultSet's that are created by queries. You have at your convenience some properties that give you the power to control whether the result set is read only, whether it can be read forward only, the direction of fetching etc.

To do this on a per query basis you would do

```
query.setHint("datanucleus.rdbms.query.fetchDirection", "forward");
query.setHint("datanucleus.rdbms.query.resultSetConcurrency", "read-only");
```

Alternatively you can specify these as persistence properties so that they apply to all queries for that PMF/EMF. Again, the properties are

- **datanucleus.rdbms.query.fetchDirection** - controls the direction that the ResultSet is navigated. By default this is forwards only. Use this property to change that.
- **datanucleus.rdbms.query.resultSetConcurrency** - controls whether the ResultSet is read only or updateable.

Bear in mind that not all RDBMS support all of the possible values for these options. That said, they do add a degree of control that is often useful.

JPQL

The JPA specification defines JPQL (a pseudo-OO query language, with SQL-like syntax), for selecting objects from the datastore. To provide a simple example, this is what you would do

```
Query q = em.createQuery("SELECT p FROM Person p WHERE p.lastName = 'Jones'");
List results = q.getResultList();
```

This finds all "Person" objects with surname of "Jones". You specify all details in the query. The *Person* specified in the query is the entity name of our entity. This defaults to the name of the class itself, but you can specify it explicitly in the mapping if wanting to use something different.

SELECT Syntax

In JPQL queries you define the query in a single string, defining the result, the candidate entity(s), the filter, any grouping, and the ordering. This string has to follow the following pattern

```
SELECT [<result>]
    FROM <from_entities_and_variables>
    [WHERE <filter>]
    [GROUP BY <grouping>] [HAVING <having>]
    [ORDER BY <ordering>]
```

The "keywords" in the query are shown in UPPER CASE, and are case-insensitive.

If you set the persistence property **datanucleus.query.jpql.allowRange** to *true* then you can optionally also specify the range of results required in the JPQL string after the ordering. It accepts the following format when this is specified

```
SELECT [<result>]
    FROM <from_entities_and_variables>
    [WHERE <filter>]
    [GROUP BY <grouping>] [HAVING <having>]
    [ORDER BY <ordering>]
    [RANGE <fromInclusive>,<toExclusive>]
```

where *fromInclusive* is the first row to be returned (origin = 0), and *toExclusive* is the row after the last one to be returned).

FROM Clause

The FROM clause declares query identification variables that represent iteration over objects in the database. The syntax of the FROM clause is as follows:

```
from_clause ::= FROM identification_variable_declaration {,
{identification_variable_declaration | collection_member_declaration}}*
identification_variable_declaration ::= range_variable_declaration { join | fetch_join
}*
range_variable_declaration ::= entity_name [AS] identification_variable

join ::= join_spec join_association_path_expression [AS] identification_variable
[join_condition]
fetch_join ::= join_spec FETCH join_association_path_expression
join_association_path_expression ::= join_collection_valued_path_expression |
join_single_valued_path_expression |
    TREAT(join_collection_valued_path_expression AS subtype) |
TREAT(join_single_valued_path_expression AS subtype)

join_collection_valued_path_expression::=
identification_variable.{single_valued_embeddable_object_field.}*collection_valued_fie
ld
join_single_valued_path_expression::=
identification_variable.{single_valued_embeddable_object_field.}*single_valued_object_
field
join_spec ::= [ LEFT [OUTER] | INNER ] JOIN
join_condition ::= ON conditional_expression
collection_member_declaration ::= IN (collection_valued_path_expression) [AS]
identification_variable
```

The FROM clause firstly defines the *candidate* entity for the query. You can specify the candidate fully-qualified, or you can specify just the **entity name**. Using our example

```
# Using candidate name fully qualified
SELECT p FROM mydomain.Person p


# Using entity name
SELECT p FROM Person p
```

By default the **entity name** is the last part of the class name (without the package), but you can specify it in metadata

Firstly, in XML metadata

```
<entity class="mydomain.Person" name="ThePerson">
    ...
</entity>
```

or using annotations

```
@Entity(name="ThePerson")
public class Person ...
```

The FROM clause also allows a user to add some explicit joins to related entities, and assign aliases to the joined entities. These are then usable in the filter/ordering/result etc. If you don't add any joins DataNucleus will add joins where they are implicit from the filter expression for example. The FROM clause is of the following structure

```
FROM {candidate_entity} {candidate_alias}
    [[[ LEFT [OUTER] | INNER ] JOIN] join_spec [join_alias] [join_condition] *
```

With JPQL you are explicitly stating that the join across *join_spec* is performed as "LEFT OUTER" or "INNER" (rather than just leaving it to DataNucleus to decide which to use). Note that the *join_spec* can be a relation field, or alternately if you have a Map of non-Entity keys/values then also the Map field. If you provide the *join_alias* then you can use it thereafter in other clauses of the query. The *join_condition* is an optional ON clause that is in addition to navigating along the relation that was specified.

Some examples of FROM clauses.

```
# Join across 2 relations, allowing referral to Address (a) and Owner (o)
SELECT p FROM Person p JOIN p.address a JOIN a.owner o WHERE o.name = 'Fred'

# Join to a Map relation field and access to the key/value of the Map.
SELECT VALUE(om) FROM Company c INNER JOIN c.officeMap om ON KEY(om) = 'London'
```

If you specify "LEFT OUTER FETCH" or "INNER FETCH" (i.e you specify **FETCH**) this means that you want those fields/properties fetching by this query. This doesn't mean that DataNucleus will definitely fetch them in the same query (because sometimes it is impossible to fetch things like multi-valued fields in a single query) but that it will attempt to fetch all fields that are selected (as well as the ones that are defaulted to EAGER).

DataNucleus JPA also allows *RIGHT* OUTER JOIN, though this is not part of the JPA spec.

FROM : Candidate that is @MappedSuperclass

In strict JPA the entity name cannot be a *MappedSuperclass* entity name. That is, if you have an abstract superclass that is persistable, you cannot query for instances of that superclass and its subclasses. We consider this a significant shortcoming of the querying capability, and allow the entity name to also be of a *MappedSuperclass*. You are unlikely to find this supported in other JPA implementations, but then maybe that's why you chose DataNucleus?

FROM : JOIN ON to another root

In strict JPA you cannot join to another "root" element. That is, you define JOIN syntax to the following element **along a relation** from the previous element. DataNucleus supports joining to a (new) "root" element potentially without any relation. See this example

```
SELECT p FROM Person p LEFT OUTER JOIN Address a ON p.addressName = a.name
```

Here we simply chose an ON clause to join the two roots.

FROM : JOIN to an embedded element

In strict JPA you cannot join to an embedded element class (of an embeddable). With DataNucleus you can do this, and hence form queries using fields of the embeddable (not available in most other JPA providers). See this example, where class *Person* has a Collection of embeddable *Address* objects.

```
SELECT p FROM Person p LEFT OUTER JOIN p.addresses a WHERE a.name = 'Home'
```

FROM : Control over INNER/OUTER join for implicit joins

RDBMS : By default if you don't specify the JOIN to some related object in the FROM clause and instead navigate through a 1-1/N-1 relation like "a.owner" then it will join using INNER JOIN. You can change this default by specifying the persistence property (to apply to all queries) or query extension **datanucleus.query.jpql.navigationJoinType** and set it to either "INNERJOIN" or "LEFTOUTERJOIN". You can also set the default for the *filter* only using the persistence property(to apply to all queries) or query extension **datanucleus.query.jpql.navigationJoinTypeForFilter** and set it to either "INNERJOIN" or "LEFTOUTERJOIN".

Fetched Fields

By default a query will fetch fields according to their defined EAGER/LAZY setting, so fields like primitives, wrappers, Dates, and 1-1/N-1 relations will be fetched, whereas 1-N/M-N fields will not be fetched. JPQL allows you to include *FETCH JOIN* as a hint to include relation fields where possible.

For RDBMS datastores any multi-valued (1-N/M-N) field (Collection, array) will be *bulk-fetched* if it is defined to be EAGER or has a *FETCH JOIN*, or is placed in the current EntityGraph. By *bulk-fetched* we mean that there will be a single SQL issued per collection field (hence avoiding the N+1 problem). By default this will be a single SQL per collection of the form `SELECT {relatedObject columns} FROM RELATED_TBL WHERE EXISTS (restrict to the candidate objects involved)`. Note that

you can disable this by either not marking multi-valued fields to be fetched, or by setting the query extension **datanucleus.rdbms.query.multivaluedFetch** to *none* (default is *exists* using the single SQL per field as mentioned above).

All non-RDBMS datastores do respect this FETCH JOIN setting, since a collection/map is stored in a single "column" in the object and so is readily retrievable.

Note that you can also make use of Entity Graphs to have fuller control over what is retrieved from each query.

WHERE clause (filter)

The most important thing to remember when defining the *filter* for JPQL is that **think how you would write it in SQL, and its likely the same except for FIELD names instead of COLUMN names**. The *filter* has to be a boolean expression, and can include the candidate entity, fields/properties, literals, functions, parameters, operators and subqueries

GROUP BY/HAVING clauses

The GROUP BY construct enables the aggregation of values according to a set of properties. The HAVING construct enables conditions to be specified that further restrict the query result. Such conditions are restrictions upon the groups. The syntax of the GROUP BY and HAVING clauses is as follows:

```
groupby_clause ::= GROUP BY groupby_item {, groupby_item}*
groupby_item ::= single_valued_path_expression | identification_variable

having_clause ::= HAVING conditional_expression
```

If a query contains both a WHERE clause and a GROUP BY clause, the effect is that of first applying the where clause, and then forming the groups and filtering them according to the HAVING clause. The HAVING clause causes those groups to be retained that satisfy the condition of the HAVING clause. The requirements for the SELECT clause when GROUP BY is used follow those of SQL: namely, any item that appears in the SELECT clause (other than as an argument to an aggregate function) must also appear in the GROUP BY clause. In forming the groups, null values are treated as the same for grouping purposes. Grouping by an entity is permitted. In this case, the entity must contain no serialized state fields or lob-valued state fields. The HAVING clause must specify search conditions over the grouping items or aggregate functions that apply to grouping items. If there is no GROUP BY clause and the HAVING clause is used, the result is treated as a single group, and the select list can only consist of aggregate functions. When a query declares a HAVING clause, it must always also declare a GROUP BY clause.

Some examples

```
SELECT p.firstName, p.lastName FROM Person p GROUP BY p.lastName

SELECT p.firstName, p.lastName FROM Person p GROUP BY p.lastName HAVING
COUNT(p.lastName) > 1
```

ORDER BY clause

The ORDER BY clause allows the objects or values that are returned by the query to be ordered. The syntax of the ORDER BY clause is

```
orderby_clause ::= ORDER BY orderby_item {, orderby_item}*
orderby_item ::= state_field_path_expression | result_variable {ASC | DESC}
```

By default your results will be returned in the order determined by the datastore, so don't rely on any particular order. You can, of course, specify the order yourself. You do this using field/property names and *ASC*/*DESC* keywords. For example

```
field1 ASC, field2 DESC
```

which will sort primarily by *field1* in ascending order, then secondarily by *field2* in descending order.

Although it is not (yet) standard JPQL, DataNucleus also supports specifying a directive for where NULL values of the ordered field/property go in the order, so the full syntax supported is

```
fieldName {ASC|DESC} {NULLS FIRST|NULLS LAST}
```

This is only supported for a few RDBMS including H2, HSQLDB, PostgreSQL, DB2, Oracle, Derby, Firebird, SQLServer v11+.

Fields/Properties

In JPQL you refer to fields/properties in the query by referring to the field/bean name. For example, if you are querying a candidate entity called *Product* and it has a field "price", then you access it like this

```
price < 150.0
```

Note that if you want to refer to a field/property of an entity you can prefix the field by its alias

```
p.price < 150.0
```

You can also chain field references if you have an entity Product (alias = p) with a field of (entity) Inventory, which has a field *name*, so you could do

```
p.inventory.name = 'Backup'
```

Note that you could alternatively have introduced a *join* to Inventory first and then just referenced the *name* field via the Inventory join alias.

Operators

The operators are listed below in order of decreasing precedence.

- Navigation operator (.)
- Arithmetic operators:
  - +, - unary
  - *, / multiplication and division
  - +, - addition and subtraction
- Comparison operators : =, >, >=, <, =<, <> (not equal), [NOT] BETWEEN, [NOT] LIKE, [NOT] IN, IS [NOT] NULL, IS [NOT] EMPTY, [NOT] MEMBER [OF], [NOT] EXISTS
- Logical operators:
  - NOT
  - AND
  - OR

Literals

JPQL supports literals of the following types : Number, boolean, character, String, *NULL* and temporal. For example, with a numeric literal

```
Query q = em.createQuery("SELECT p FROM Person p WHERE p.age = 25");
```

When String literals are specified using single format JPQL they should be surrounded by single-quotes '. For example

```
Query q = em.createQuery("SELECT p FROM Person p WHERE p.firstName = 'John'");
```

When temporal literals are specified using string format JPQL they use *JDBC escape syntax* (see the

JDBC spec for full details), namely

```
{d 'yyyy-mm-dd'}                - a Date
{t 'hh:mm:ss'}                  - a Time
{ts 'yyyy-mm-dd hh:mm:ss.f...'}  - a Timestamp
```

For example

```
Query q = em.createQuery("SELECT p FROM Person p WHERE p.birthDate < {ts '1970-01-01
00:00:00.000000001'}");
```

RDBMS : Parameters .v. Literals

When considering whether to embody a literal into a JPQL query, you should consider using a parameter instead. The advantage of using a parameter is that the generated SQL will have a '?' rather than the value. As a result, if you are using a connection pool that supports PreparedStatement caching, this will potentially reuse an existing statement rather than generating a new one each time. If you only ever invoke a query with a single possible value of the parameter then there is no advantage. If you invoke the query with multiple possible values of the parameter then this advantage can be significant.

Parameters

In JPQL queries it is convenient to pass in parameters so we don't have to define the same query for different values. Let's take two examples

```
# Named Parameters :
Query q = em.createQuery("SELECT p FROM Person p WHERE p.lastName = :surname AND
p.firstName = :forename");
q.setParameter("surname", theSurname);
q.setParameter("forename", theForename);

# Numbered Parameters :
Query q = em.createQuery("SELECT p FROM Person p WHERE p.lastName = ?1 AND p.firstName
= ?2");
q.setParameter(1, theSurname);
q.setParameter(2, theForename);
```

So in the first case we have parameters that are prefixed by **:** (colon) to identify them as a parameter and we use that name when calling *Query.setParameter()*. In the second case we have parameters that are prefixed by **?** (question mark) and are numbered starting at 1. We then use the numbered position when calling *Query.setParameter()*.

CASE expressions

For particular use in the *result* clause, you can make use of a **CASE** expression where you want to return different things based on some condition(s). Like this

```
Query q = em.createQuery(
     "SELECT p.personNum, CASE WHEN p.age < 18 THEN 'Youth' WHEN p.age >= 18 AND p.age
< 65 THEN 'Adult' ELSE 'Old' END FROM Person p");
```

So in this case the second result value will be a String, either "Youth", "Adult" or "Old" depending on the age of the person. The BNF structure of the JPQL CASE expression is

```
CASE WHEN conditional_expression THEN scalar_expression
    {WHEN conditional_expression THEN scalar_expression}*
    ELSE scalar_expression
END
```

JPQL Functions

JPQL provides an SQL-like query language. Just as with SQL, JPQL also supports a range of functions to enhance the querying possibilities. The tables below also mark whether a particular method is supported for evaluation in-memory.

These methods are not available for use with all of the supported datastores to be executed in-datastore. RDBMS, in general, supports the vast majority, whilst MongoDB, Neo4j, Cassandra support a select few methods in-datastore.

Please note that you can easily add support for other functions for evaluation "in-memory" using this DataNucleus plugin point

Please note that you can easily add support for other functions with RDBMS datastore using this DataNucleus plugin point

Aggregate Functions

There are a series of aggregate functions for aggregating the values of a field for all rows of the results.

Function Name	Description	Standard	In-Memory
COUNT(field)	Returns the aggregate count of the field (Long)	✔	✔

Function Name	Description	Standard	In-Memory
MIN(field)	Returns the minimum value of the field (type of the field)	✔	✔
MAX(field)	Returns the maximum value of the field (type of the field)	✔	✔
AVG(field)	Returns the average value of the field (Double)	✔	✔
SUM(field)	Returns the sum of the field value(s) (Long, Double, BigInteger, BigDecimal)	✔	✔

String Functions

There are a series of functions to be applied to String fields.

Function Name	Description	Standard	In-Memory
CONCAT(str_field, str_field2 [, str_fieldX])	Returns the concatenation of the string fields	✔	✔
SUBSTRING(str_field, num1 [, num2])	Returns the substring of the string field starting at position *num1*, and optionally with the length of *num2*	✔	✔
TRIM([trim_spec] [trim_char] [FROM] str_field)	Returns trimmed form of the string field	✔	✔
LOWER(str_field)	Returns the lower case form of the string field	✔	✔
UPPER(str_field)	Returns the upper case form of the string field	✔	✔
LENGTH(str_field)	Returns the size of the string field (number of characters)	✔	✔
LOCATE(str_field1, str_field2 [, num])	Returns position of *str_field2* in *str_field1* optionally starting at *num*	✔	✔

Temporal Functions

There are a series of functions for use with temporal values

Function Name	Description	Standard	In-Memory
CURRENT_DATE	Returns the current date (day month year) of the datastore server	✔	✔

Function Name	Description	Standard	In-Memory
CURRENT_TIME	Returns the current time (hour minute second) of the datastore server	✔	✔
CURRENT_TIMESTAMP	Returns the current timestamp of the datastore server	✔	✔
YEAR(dateField)	Returns the year of the specified date in timezone it was stored	✖	✔
MONTH(dateField)	Returns the month of the specified date (1-12) in timezone it was stored.	✖	✔
MONTH_JAVA(dateField)	Returns the month of the specified date (0-11) in timezone it was stored	✖	✔
DAY(dateField)	Returns the day of the month of the specified date in timezone it was stored	✖	✔
DAY_OF_WEEK(dateField)	Returns the day of the week of the specified date in timezone it was stored (1-7, with sunday as 1)	✖	✔
HOUR(dateField)	Returns the hour of the specified date in timezone it was stored	✖	✔
MINUTE(dateField)	Returns the minute of the specified date in timezone it was stored	✖	✔
SECOND(dateField)	Returns the second of the specified date in timezone it was stored	✖	✔

Collection Functions

There are a series of functions for use with collection values

Function Name	Description	Standard	In-Memory
INDEX(collection_field)	Returns index number of the field element when that is the element of an indexed List field.	✔	✖
SIZE(collection_field)	Returns the size of the collection field. Empty collection will return 0	✔	✔

Map Functions

There are a series of functions for use with maps

Function Name	Description	Standard	In-Memory
KEY(map_field)	Returns the key of the map	✔	✖
VALUE(map_field)	Returns the value of the map	✔	✔
SIZE(map_field)	Returns the size of the map field. Empty map will return 0	✔	✔

Arithmetic Functions

There are a series of functions for arithmetic use

Function Name	Description	Standard	In-Memory
ABS(numeric_field)	Returns the absolute value of the numeric field	✔	✔
SQRT(numeric_field)	Returns the square root of the numeric field	✔	✔
MOD(num_field1, num_field2)	Returns the modulus of the two numeric fields (*num_field1 % num_field2)*	✔	✔
ACOS(num_field)	Returns the arc-cosine of a numeric field	✖	✔
ASIN(num_field)	Returns the arc-sine of a numeric field	✖	✔
ATAN(num_field)	Returns the arc-tangent of a numeric field	✖	✔
COS(num_field)	Returns the cosine of a numeric field	✖	✔
SIN(num_field)	Returns the sine of a numeric field	✖	✔
TAN(num_field)	Returns the tangent of a numeric field	✖	✔
DEGREES(num_field)	Returns the degrees of a numeric field	✖	✔
RADIANS(num_field)	Returns the radians of a numeric field	✖	✔
CEIL(num_field)	Returns the ceiling of a numeric field	✖	✔
FLOOR(num_field)	Returns the floor of a numeric field	✖	✔
LOG(num_field)	Returns the natural logarithm of a numeric field	✖	✔
EXP(num_field)	Returns the exponent of a numeric field	✖	✔
POWER(numeric_field, numeric_value)	Returns the numeric field to the specified power	✖	✖

Other Functions

You have a further function available

Function Name	Description	Standard	In-Memory
FUNCTION(name, [arg1 [,arg2 ...]])	Executes the specified SQL function "name" with the defined arguments. **RDBMS only**	✔	✖

For example, this executes the SQL function 'date_part' (where it is available) with 2 arguments, a Date, and a format. Clearly there are better ways of handling dates than this so it serves simply as an example of invocation

```
SELECT FUNCTION('date_part', myDate, 'YYYY-MM-DD') FROM ...
```

In addition, DataNucleus JPA provides support for a number of Geospatial functions.

Collection Fields

Where you have a collection field, often you want to navigate it to query based on some filter for the element. To achieve this, you can clearly JOIN to the element in the FROM clause. Alternatively you can use the *MEMBER OF* keyword. Let's take an example, you have a field which is a Collection of Strings, and want to return the owner object that has an element that is *"Freddie"*.

```
Query q = em.createQuery("SELECT p.firstName, p.lastName FROM Person p WHERE 'Freddie'
MEMBER OF p.nicknames");
```

Beyond this, you can also make use of the collection functions and use the size of the collection for example.

Map Fields

Where you have a map field, often you want to navigate it to query based on some filter for the key or value. Let's take an example, you want to return the value for a particular key in the map of an owner.

```
Query q = em.createQuery("SELECT VALUE(p.addresses) FROM Person p WHERE
KEY(p.addresses) = 'London Flat'");
```

Beyond this, you can also make use of the map functions and use the size of the map for example.

in the JPA spec they allow a user to interchangeably use "p.addresses" to refer to the *value* of the Map. Whilst DataNucleus supports this, we advise using explicit *VALUE({field})* since it is clearer the intent and makes for more readable queries.

Subqueries

In strict JPQL you can only have subqueries in WHERE or HAVING clauses. DataNucleus additionally allows them in the SELECT, GROUP and ORDER clauses.

With JPQL the user has a very flexible query syntax which allows for querying of the vast majority of data components in a single query. In some situations it is desirable for the query to utilise the results of a separate query in its calculations. JPQL also allows the use of subqueries. Here's an example

```
SELECT e FROM Employee e
WHERE e.salary > (SELECT avg(f.salary) FROM Employee f)
```

So we want to find all Employees that have a salary greater than the average salary. The subquery must be in parentheses (brackets). Note that we have defined the subquery with an alias of "f", whereas in the outer query the alias is "e".

ALL/ANY/SOME Expressions

One use of subqueries with JPQL is where you want to compare with some or all of a particular expression. To give an example

```
SELECT emp FROM Employee emp
WHERE emp.salary > ALL (SELECT m.salary FROM Manager m WHERE m.department =
emp.department)
```

So this returns all employees that earn more than all managers in the same department! You can also compare with SOME/ANY, like this

```
SELECT emp FROM Employee emp
WHERE emp.salary > ANY (SELECT m.salary FROM Manager m WHERE m.department =
emp.department)
```

So this returns all employees that earn more than any one Manager in the same department.

EXISTS Expressions

Another use of subqueries in JPQL is where you want to check on the existence of a particular thing. For example

```
SELECT DISTINCT emp FROM Employee emp
WHERE EXISTS (SELECT emp2 FROM Employee emp2 WHERE emp2 = emp.spouse)
```

So this returns the employees that have a partner also employed.

Specify candidates to query over

With JPA you always query objects of the candidate type in the datastore. DataNucleus extends this and allows you to provide a Collection of candidate objects that will be queried (rather than going to the datastore), and it will perform the querying "in-memory". You set the candidates like this

```
Query query = em.createQuery("SELECT p FROM Products p WHERE ...");
((org.datanucleus.api.jpa.JPAQuery)query).getInternalQuery().setCandidates(myCandidate
s);
List<Product> results = query.getResultList();
```

Range of Results

With JPQL you can select the range of results to be returned. For example if you have a web page and you are paginating the results of some search, you may want to get the results from a query in blocks of 20 say, with results 0 to 19 on the first page, then 20 to 39, etc. You can facilitate this as follows

```
Query q = em.createQuery("SELECT p FROM Person p WHERE p.age > 20");
q.setFirstResult(0);
q.setMaxResults(20);
```

So with this query we get results 0 to 19 inclusive.

Query Result

Whilst the majority of the time you will want to return instances of a candidate class, JPQL also allows you to return customised results. Consider the following example

```
Query q = em.createQuery("SELECT p.firstName, p.lastName FROM Person p WHERE p.age >
20");
List<Object[]> results = q.getResultList();
```

this returns the first and last name for each Person meeting that filter. Obviously we may have some container class that we would like the results returned in, so if we change the query to this

```
Query<PersonName> q = em.createQuery(
    "SELECT p.firstName, p.lastName FROM Person p WHERE p.age > 20", PersonName.
class);
List<PersonName> results = q.getResultList();
```

so each result is a PersonName, holding the first and last name. This result class needs to match one

of the following structures

- Constructor taking arguments of the same types and the same order as the result clause. An instance of the result class is created using this constructor. For example

```
public class PersonName
{
    protected String firstName = null;
    protected String lastName = null;
    public PersonName(String first, String last)
    {
        this.firstName = first;
        this.lastName = last;
    }
}
```

- Default constructor, and setters for the different result columns, using the alias name for each column as the property name of the setter. For example

```
public class PersonName
{
    protected String firstName = null;
    protected String lastName = null;
    public PersonName()
    {
    }
    public void setFirstName(String first) {this.firstName = first;}
    public void setLastName(String last) {this.lastName = last;}
}
```

- Default constructor, and a method *void put(Object aliasName, Object value)*

Note that if the setter property name doesn't match the query result component name, you should use *AS {alias}* in the query so they are the same.

Tuples

A special case, where you don't have a result class but want to easily extract multiple columns in the form of a **Tuple** JPA provides a special class *javax.persistence.Tuple* to supply as the result class in the above call. From that you can get hold of the column aliases, and their values.

```
Query<PersonName> q = em.createQuery(
    "SELECT p.firstName, p.lastName FROM Person p WHERE p.age > 20", Tuple.class);
List<Tuple> results = q.getResultList();
for (Tuple t : results)
{
    List<TupleElement> cols = t.getElements();
    for (TupleElement col : cols)
    {
        String colName = col.getAlias();
        Object value = t.get(colname);
    }
}
```

Query Execution

There are two ways to execute a JPQL query. When you know it will return 0 or 1 results you call

```
Object result = query.getSingleResult();
```

If however you know that the query will return multiple results, or you just don't know then you would call

```
List results = query.getResultList();
```

When using RDBMS all parts of a query are evaluated **in-datastore**. When using LDAP, Excel, ODF, XML, JSON, GoogleStorage, AmazonS3 any query filter/ordering etc is evaluated **in-memory**. When using Neo4j, HBase, MongoDB and Cassandra any query filter/ordering etc are evaluated **in-datastore** where possible, with the remainder evaluated **in-memory**.

JPQL In-Memory queries

The typical use of a JPQL query is to translate it into the native query language of the datastore and return objects matched by the query. For many (usually non-RDBMS) datastores it is simply impossible to support the full JPQL syntax in the datastore *native query language* and so it is necessary to evaluate the query in-memory. This means that we evaluate as much as we can in the datastore and then instantiate those objects and evaluate further in-memory. Here we document the current capabilities of *in-memory evaluation* in DataNucleus.

- Subqueries using ALL, ANY, SOME, EXISTS are not currently supported for use in-memory.
- MEMBER OF syntax is not currently supported for use in-memory.

To enable evaluation in memory you specify the query hint **datanucleus.query.evaluateInMemory** to *true* as follows

```
query.setHint("datanucleus.query.evaluateInMemory","true");
```

In-memory JPQL evaluation does not support JOINs currently, or correlated subqueries. You should omit such things from your query and try to evaluate them manually in your own code.

Named Query

With the JPA API you can either define a query at runtime, or define it in the MetaData/annotations for a class and refer to it at runtime using a symbolic name. This second option means that the method of invoking the query at runtime is much simplified. To demonstrate the process, lets say we have a class called *Product* (something to sell in a store). We define the JPA Meta-Data for the class in the normal way, but we also have some query that we know we will require, so we define the following in the Meta-Data.

```
<entity class="Product">
    ...
    <named-query name="SoldOut"><![CDATA[
    SELECT p FROM Product p WHERE p.status = "Sold Out"
    ]]></named-query>
</entity>
```

or using annotations

```
@Entity
@NamedQuery(name="SoldOut", query="SELECT p FROM Product p WHERE p.status = 'Sold
Out'")
public class Product {...}
```

DataNucleus also supports specifying this using `@NamedQuery` annotation in non-Entity classes. This is beyond the JPA spec, but is very useful in real applications.

Above we have a JPQL query called "SoldOut" defined for the class *Product* that returns all Products (and subclasses) that have a *status* of "Sold Out". To execute this query we would do as follows

```
Query query = em.createNamedQuery("SoldOut");
List<Product> results = query.getResultList();
```

Saving a Query as a Named Query

You can save a query as a named query like this

```
Query q = em.createQuery("SELECT p FROM Product p WHERE ...");
...
emf.addNamedQuery("MyQuery", q);
```

DataNucleus also allows you to create a query, and then save it as a "named" query directly with the query. You do this as follows

```
Query q = em.createQuery("SELECT p FROM Product p WHERE ...");
((org.datanucleus.api.jpa.JPAQuery)q).saveAsNamedQuery("MyQuery");
```

With both methods you can thereafter access the query via

```
Query q = em.createNamedQuery("MyQuery");
```

JPQL Strictness

By default DataNucleus allows some extensions in syntax over strict JPQL (as defined by the JPA spec). To allow only strict JPQL you can do as follows

```
Query query = em.createQuery(...);
query.setHint("datanucleus.jpql.strict", "true");
```

JPQL : SQL Generation for RDBMS

With a JPQL query running on an RDBMS the query is compiled into SQL. Here we give a few examples of what SQL is generated. You can of course try this for yourself observing the content of the DataNucleus log, or by using the following vendor extension

```
Query q = em.createQuery(...);
List results = q.getResultList();

String sql = (String)((org.datanucleus.api.jpa.JPAQuery)q).getNativeQuery();
```

For non-RDBMS datastores this can return other object types.

In JPQL you specify a candidate class and its alias (identifier). In addition you can specify joins with their respective alias. The DataNucleus implementation of JPQL will preserve these aliases in the

generated SQL.

```
# JPQL:
SELECT p FROM Person p INNER JOIN p.bestFriend AS B

# SQL:
SELECT P.ID
FROM PERSON P INNER JOIN PERSON B ON B.ID = P.BESTFRIEND_ID
```

With the JPQL *MEMBER OF* syntax this is typically converted into an EXISTS query.

```
# JPQL:
SELECT DISTINCT p FROM Person p WHERE :param MEMBER OF p.friends

# SQL:
SELECT DISTINCT P.ID FROM PERSON P
WHERE EXISTS (
    SELECT 1 FROM PERSON_FRIENDS P_FRIENDS, PERSON P_FRIENDS_1
    WHERE P_FRIENDS.PERSON_ID = P.ID
    AND P_FRIENDS_1.GLOBAL_ID = P_FRIENDS.FRIEND_ID
    AND 101 = P_FRIENDS_1.ID)
```

JPQL DELETE Queries

The JPA specification defines a mode of JPQL for deleting objects from the datastore. NOTE: his will not invoke any cascading defined on a field basis, with only datastore-defined Foreign Keys cascading. Additionally related objects already in-memory will not be updated.

DELETE Syntax

The syntax for deleting records is very similar to selecting them

```
DELETE FROM [<candidate-class> [[AS] {alias}]]  [WHERE <filter>]
```

The "keywords" in the query are shown in UPPER CASE, and are case-insensitive.

```
Query query = em.createQuery("DELETE FROM Person p WHERE firstName = 'Fred'");
int numRowsDeleted = query.executeUpdate();
```

JPQL UPDATE Queries

The JPA specification defines a mode of JPQL for updating objects in the datastore.

This will not invoke any cascading defined on a field basis, with only datastore-defined Foreign Keys cascading. Additionally related objects already in-memory will not be updated

UPDATE Syntax

The syntax for updating records is very similar to selecting them

```
UPDATE [<candidate-class> [[AS] {alias}]] SET item1=value1, item2=value2  [WHERE
<filter>]
```

The "keywords" in the query are shown in UPPER CASE, and are case-insensitive.

```
Query query = em.createQuery("UPDATE Person p SET p.salary = 10000 WHERE age = 18");
int numRowsUpdated = query.executeUpdate();
```

In strict JPA you cannot use a subquery in the UPDATE clause. With DataNucleus JPA you can do this so, for example, you can set a field to the result of a subquery.

```
Query query = em.createQuery("UPDATE Person p SET p.salary = (SELECT MAX(p2.salary)
FROM Person p2 WHERE age < 18) WHERE age = 18");
```

JPQL BNF Notation

The BNF defining the JPQL query language is shown below.

```
QL_statement ::= select_statement | update_statement | delete_statement
select_statement ::= select_clause from_clause [where_clause] [groupby_clause]
[having_clause] [orderby_clause]

update_statement ::= update_clause [where_clause]

delete_statement ::= delete_clause [where_clause]

from_clause ::= FROM identification_variable_declaration
    {, {identification_variable_declaration | collection_member_declaration}}*
identification_variable_declaration ::= range_variable_declaration { join | fetch_join
}*
range_variable_declaration ::= entity_name [AS] identification_variable

join ::= join_spec join_association_path_expression [AS] identification_variable
fetch_join ::= join_spec FETCH join_association_path_expression
join_spec::= [ LEFT [OUTER] | INNER ] JOIN
```

```
join_association_path_expression ::= join_collection_valued_path_expression |
join_single_valued_path_expression
join_collection_valued_path_expression::=

identification_variable.{single_valued_embeddable_object_field.}*collection_valued_fie
ld
join_single_valued_path_expression::=

identification_variable.{single_valued_embeddable_object_field.}*single_valued_object_
field
collection_member_declaration ::=
    IN (collection_valued_path_expression) [AS] identification_variable
qualified_identification_variable ::= KEY(identification_variable) |
VALUE(identification_variable) |
    ENTRY(identification_variable)
single_valued_path_expression ::= qualified_identification_variable |
    state_field_path_expression | single_valued_object_path_expression
general_identification_variable ::= identification_variable |
KEY(identification_variable) |
    VALUE(identification_variable)

state_field_path_expression ::=
general_identification_variable.{single_valued_object_field.}*state_field
single_valued_object_path_expression ::=
    general_identification_variable.{single_valued_object_field.}*
single_valued_object_field
collection_valued_path_expression ::=

general_identification_variable.{single_valued_object_field.}*collection_valued_field

update_clause ::= UPDATE entity_name [[AS] identification_variable] SET update_item {,
update_item}*
update_item ::= [identification_variable.]{state_field | single_valued_object_field} =
new_value
new_value ::= scalar_expression | simple_entity_expression | NULL

delete_clause ::= DELETE FROM entity_name [[AS] identification_variable]

select_clause ::= SELECT [DISTINCT] select_item {, select_item}*
select_item ::= select_expression [[AS] result_variable]
select_expression ::= single_valued_path_expression | scalar_expression |
aggregate_expression |
    identification_variable | OBJECT(identification_variable) | constructor_expression
constructor_expression ::= NEW constructor_name ( constructor_item {,
constructor_item}* )
constructor_item ::= single_valued_path_expression | scalar_expression |
aggregate_expression |
    identification_variable

aggregate_expression ::= { AVG | MAX | MIN | SUM } ([DISTINCT]
state_field_path_expression) |
```

```
    COUNT ([DISTINCT] identification_variable | state_field_path_expression |
    single_valued_object_path_expression)

where_clause ::= WHERE conditional_expression
groupby_clause ::= GROUP BY groupby_item {, groupby_item}*
groupby_item ::= single_valued_path_expression | identification_variable
having_clause ::= HAVING conditional_expression
orderby_clause ::= ORDER BY orderby_item {, orderby_item}*
orderby_item ::= state_field_path_expression | result_variable [ ASC | DESC ]

subquery ::= simple_select_clause subquery_from_clause [where_clause] [groupby_clause]
[having_clause]
subquery_from_clause ::= FROM subselect_identification_variable_declaration
    {, subselect_identification_variable_declaration | collection_member_declaration}*

subselect_identification_variable_declaration ::= identification_variable_declaration
|
    derived_path_expression [AS] identification_variable {join}*|
    derived_collection_member_declaration
derived_path_expression ::=

superquery_identification_variable.{single_valued_object_field.}*collection_valued_fie
ld |

superquery_identification_variable.{single_valued_object_field.}*single_valued_object_
field
derived_collection_member_declaration ::=
    IN
superquery_identification_variable.{single_valued_object_field.}*collection_valued_fie
ld
simple_select_clause ::= SELECT [DISTINCT] simple_select_expression
simple_select_expression::= single_valued_path_expression | scalar_expression |
aggregate_expression |
    identification_variable
scalar_expression ::= simple_arithmetic_expression | string_primary | enum_primary |
    datetime_primary | boolean_primary | case_expression | entity_type_expression
conditional_expression ::= conditional_term | conditional_expression OR
conditional_term
conditional_term ::= conditional_factor | conditional_term AND conditional_factor
conditional_factor ::= [ NOT ] conditional_primary
conditional_primary ::= simple_cond_expression | (conditional_expression)
simple_cond_expression ::= comparison_expression | between_expression |
    in_expression | like_expression | null_comparison_expression |
    empty_collection_comparison_expression | collection_member_expression |
exists_expression
between_expression ::=
    arithmetic_expression [NOT] BETWEEN arithmetic_expression AND
arithmetic_expression |
    string_expression [NOT] BETWEEN string_expression AND string_expression |
    datetime_expression [NOT] BETWEEN datetime_expression AND datetime_expression
in_expression ::= {state_field_path_expression | type_discriminator} [NOT] IN
```

```
    { ( in_item {, in_item}* ) | (subquery) | collection_valued_input_parameter }
in_item ::= literal | single_valued_input_parameter
like_expression ::= string_expression [NOT] LIKE pattern_value [ESCAPE
escape_character]
null_comparison_expression ::= {single_valued_path_expression | input_parameter} IS
[NOT] NULL

empty_collection_comparison_expression ::= collection_valued_path_expression IS [NOT]
EMPTY
collection_member_expression ::= entity_or_value_expression [NOT] MEMBER [OF]
collection_valued_path_expression
entity_or_value_expression ::= single_valued_object_path_expression |
state_field_path_expression |
    simple_entity_or_value_expression
simple_entity_or_value_expression ::= identification_variable | input_parameter |
literal
exists_expression::= [NOT] EXISTS (subquery)
all_or_any_expression ::= { ALL | ANY | SOME} (subquery)
comparison_expression ::=
    string_expression comparison_operator {string_expression | all_or_any_expression}
|
    boolean_expression { =|<>} {boolean_expression | all_or_any_expression} |
    enum_expression { =|<>} {enum_expression | all_or_any_expression} |
    datetime_expression comparison_operator
    {datetime_expression | all_or_any_expression} |
    entity_expression { = | <>} {entity_expression | all_or_any_expression} |
    arithmetic_expression comparison_operator
    {arithmetic_expression | all_or_any_expression} |
    entity_type_expression { =|<>} entity_type_expression}
comparison_operator ::= = | > | >= | < | <= | <>
arithmetic_expression ::= simple_arithmetic_expression | (subquery)
simple_arithmetic_expression ::= arithmetic_term | simple_arithmetic_expression { + |
- } arithmetic_term
arithmetic_term ::= arithmetic_factor | arithmetic_term { * | / } arithmetic_factor
arithmetic_factor ::= [{ + | - }] arithmetic_primary
arithmetic_primary ::= state_field_path_expression | numeric_literal |
    (simple_arithmetic_expression) | input_parameter | functions_returning_numerics |
     aggregate_expression | case_expression
string_expression ::= string_primary | (subquery)
string_primary ::= state_field_path_expression | string_literal |
    input_parameter | functions_returning_strings | aggregate_expression |
case_expression
datetime_expression ::= datetime_primary | (subquery)

datetime_primary ::= state_field_path_expression | input_parameter |
functions_returning_datetime |
    aggregate_expression | case_expression | date_time_timestamp_literal
boolean_expression ::= boolean_primary | (subquery)
boolean_primary ::= state_field_path_expression | boolean_literal | input_parameter |
    case_expression
enum_expression ::= enum_primary | (subquery)
```

```
enum_primary ::= state_field_path_expression | enum_literal | input_parameter |
case_expression
entity_expression ::= single_valued_object_path_expression | simple_entity_expression
simple_entity_expression ::= identification_variable | input_parameter
entity_type_expression ::= type_discriminator | entity_type_literal | input_parameter
type_discriminator ::= TYPE(identification_variable |
single_valued_object_path_expression |
    input_parameter)
functions_returning_numerics::= LENGTH(string_primary) |
    LOCATE(string_primary, string_primary[, simple_arithmetic_expression]) |
    ABS(simple_arithmetic_expression) |
    SQRT(simple_arithmetic_expression) |
    MOD(simple_arithmetic_expression, simple_arithmetic_expression) |
    SIZE(collection_valued_path_expression) |
    INDEX(identification_variable)
functions_returning_datetime ::= CURRENT_DATE | CURRENT_TIME | CURRENT_TIMESTAMP

functions_returning_strings ::=
    CONCAT(string_primary, string_primary {, string_primary}*) |
    SUBSTRING(string_primary, simple_arithmetic_expression [,
simple_arithmetic_expression]) |
    TRIM([[trim_specification] [trim_character] FROM] string_primary) |
    LOWER(string_primary) |
    UPPER(string_primary)
trim_specification ::= LEADING | TRAILING | BOTH
case_expression ::= general_case_expression | simple_case_expression |
coalesce_expression |
    nullif_expression
general_case_expression::= CASE when_clause {when_clause}* ELSE scalar_expression END
when_clause::= WHEN conditional_expression THEN scalar_expression
simple_case_expression::=
    CASE case_operand simple_when_clause {simple_when_clause}*
    ELSE scalar_expression
    END
case_operand::= state_field_path_expression | type_discriminator
simple_when_clause::= WHEN scalar_expression THEN scalar_expression
coalesce_expression::= COALESCE(scalar_expression {, scalar_expression}+)
nullif_expression::= NULLIF(scalar_expression, scalar_expression)
```

Geospatial Functions

When querying spatial data you can make use of a set of spatial methods on the various Java geometry types. The list contains all of the functions detailed in Section 3.2 of the OGC Simple Features specification. Additionally DataNucleus provides some commonly required methods like bounding box test and datastore specific functions. The following tables list all available functions as well as information about which RDBMS implement them. An entry in the "Result" column indicates, whether the funcion may be used in the result part of a JPQL query.

Functions for Constructing a Geometry Value given its Well-known Text Representation (OGC SF 3.2.6)

Method	Description	Specification	Result [1]	PostGIS	MySQL	Oracle Spatial
Spatial.geomFromText(String, Integer)	Construct a Geometry value given its well-known textual representation.	OGC SF	✖	✔	✔	✔
Spatial.pointFromText(String, Integer)	Construct a Point.	OGC SF	✖	✔	✔	✔
Spatial.lineFromText(String, Integer)	Construct a LineString.	OGC SF	✖	✔	✔	✔
Spatial.polyFromText(String, Integer)	Construct a Polygon.	OGC SF	✖	✔	✔	✔
Spatial.mPointFromText(String, Integer)	Construct a MultiPoint.	OGC SF	✖	✔	✔	✔
Spatial.mLineFromText(String, Integer)	Construct a MultiLineString.	OGC SF	✖	✔	✔	✔
Spatial.mPolyFromText(String, Integer)	Construct a MultiPolygon.	OGC SF	✖	✔	✔	✔
Spatial.geomCollFromText(String, Integer)	Construct a GeometryCollection.	OGC SF	✖	✔	✔	✔

[1] These functions can't be used in the return part because it's not possible to determine the return type from the parameters.

Functions for Constructing a Geometry Value given its Well-known Binary Representation (OGC SF 3.2.7)

Method	Description	Specification	Result [1]	PostGIS	MySQL	Oracle Spatial
Spatial.geomFromWKB(String, Integer)	Construct a Geometry value given its well-known binary representation.	OGC SF	✖	✔	✔	✔
Spatial.pointFromWKB(String, Integer)	Construct a Point.	OGC SF	✖	✔	✔	✔
Spatial.lineFromWKB(String, Integer)	Construct a LineString.	OGC SF	✖	✔	✔	✔

Method	Description	Specification	Result [1]	PostGIS	MySQL	Oracle Spatial
Spatial.polyFromWKB(String, Integer)	Construct a Polygon.	OGC SF	✖	✔	✔	✔
Spatial.mPointFromWKB(String, Integer)	Construct a MultiPoint.	OGC SF	✖	✔	✔	✔
Spatial.mLineFromWKB(String, Integer)	Construct a MultiLineString.	OGC SF	✖	✔	✔	✔
Spatial.mPolyFromWKB(String, Integer)	Construct a MultiPolygon.	OGC SF	✖	✔	✔	✔
Spatial.geomCollFromWKB(String, Integer)	Construct a GeometryCollection.	OGC SF	✖	✔	✔	✔

[1] These functions can't be used in the return part because it's not possible to determine the return type from the parameters.

Functions on Type Geometry (OGC SF 3.2.10)

Method	Description	Specification	Result [1]	PostGIS	MySQL	Oracle Spatial
Spatial.dimension(Geometry)	Returns the dimension of the Geometry.	OGC SF	✔	✔	✔	✔
Spatial.geometryType(Geometry)	Returns the name of the instantiable subtype of Geometry.	OGC SF	✔	✔	✔	✔
Spatial.asText(Geometry)	Returns the well-known textual representation.	OGC SF	✔	✔	✔	✔
Spatial.asBinary(Geometry)	Returns the well-known binary representation.	OGC SF	✖	✔	✔	✔
Spatial.srid(Geometry)	Returns the Spatial Reference System ID for this Geometry.	OGC SF	✔	✔	✔	✔
Spatial.isEmpty(Geometry)	TRUE if this Geometry corresponds to the empty set.	OGC SF	! [1]	✔	✔	✔
Spatial.isSimple(Geometry)	TRUE if this Geometry is simple, as defined in the Geometry Model.	OGC SF	! [1]	✔	✔	✔
Spatial.boundary(Geometry)	Returns a Geometry that is the combinatorial boundary of the Geometry.	OGC SF	✔	✔	✔	✔

Method	Description	Specification	Result [1]	PostGIS	MySQL	Oracle Spatial
Spatial.envelope(Geometry)	Returns the rectangle bounding Geometry as a Polygon.	OGC SF	✔	✔	✔	✔

[1] Oracle does not allow boolean expressions in the SELECT-list.

Functions on Type Point (OGC SF 3.2.11)

Method	Description	Specification	Result [1]	PostGIS	MySQL	Oracle Spatial
Spatial.x(Point)	Returns the x-coordinate of the Point as a Double.	OGC SF	✔	✔	✔	✔
Spatial.y(Point)	Returns the y-coordinate of the Point as a Double.	OGC SF	✔	✔	✔	✔

Functions on Type Curve (OGC SF 3.2.12)

Method	Description	Specification	Result [1]	PostGIS	MySQL	Oracle Spatial
Spatial.startPoint(Curve))	Returns the first point of the Curve.	OGC SF	✔	✔	✔	✔
Spatial.endPoint(Curve))	Returns the last point of the Curve.	OGC SF	✔	✔	✔	✔
Spatial.isRing(Curve)	Returns TRUE if Curve is closed and simple. .	OGC SF	! [1]	✔	✔	✔

[1] Oracle does not allow boolean expressions in the SELECT-list.

Functions on Type Curve and Type MultiCurve (OGC SF 3.2.12, 3.2.17)

Method	Description	Specification	Result [1]	PostGIS	MySQL	Oracle Spatial
Spatial.isClosed(Curve), Spatial.isClosed(MultiCurve)	Returns TRUE if Curve is closed, i.e., if StartPoint(Curve) = EndPoint(Curve).	OGC SF	! [1]	✔	✔	✔
Spatial.length(Curve), Spatial.length(MultiCurve)	Returns the length of the Curve.	OGC SF	✔	✔	✔	✔

[1] Oracle does not allow boolean expressions in the SELECT-list.

Functions on Type LineString (OGC SF 3.2.13)

Method	Description	Specification	Result [1]	PostGIS	MySQL	Oracle Spatial
Spatial.numPoints(LineString)	Returns the number of points in the LineString.	OGC SF	✔	✔	✔	✔
Spatial.pointN(LineString, Integer)	Returns Point n.	OGC SF	✔	✔	✔	✔

Functions on Type Surface and Type MultiSurface (OGC SF 3.2.14, 3.2.18)

Method	Description	Specification	Result [1]	PostGIS	MySQL	Oracle Spatial
Spatial.centroid(Surface), centroid(MultiSurface)	Returns the centroid of Surface, which may lie outside of it.	OGC SF	✔	✔	✖ [1]	✔
Spatial.pointOnSurface(Surface), pointOnSurface(MultiSurface)	Returns a Point guaranteed to lie on the surface.	OGC SF	✔	✔	✖ [1]	✔
Spatial.area(Surface), area(MultiSurface)	Returns the area of Surface.	OGC SF	✔	✔	✔	✔

[1] MySQL does not implement these functions.

Functions on Type Polygon (OGC SF 3.2.15)

Method	Description	Specification	Result [1]	PostGIS	MySQL	Oracle Spatial
Spatial.exteriorRing(Polygon)	Returns the exterior ring of Polygon.	OGC SF	✔	✔	✔	✔
Spatial.numInteriorRing(Polygon)	Returns the number of interior rings.	OGC SF	✔	✔	✔	✔
Spatial.interiorRingN(Polygon, Integer)	Returns the nth interior ring.	OGC SF	✔	✔	✔	✔

Functions on Type GeomCollection (OGC SF 3.2.16)

Method	Description	Specification	Result [1]	PostGIS	MySQL	Oracle Spatial
Spatial.numGeometries(GeomCollection)	Returns the number of geometries in the collection.	OGC SF	✔	✔	✔	✔
Spatial.geometryN(GeomCollection, Integer)	Returns the nth geometry in the collection.	OGC SF	✔	✔	✔	✔

Functions that test Spatial Relationships (OGC SF 3.2.19)

Method	Description	Specification	Result [1]	PostGIS	MySQL	Oracle Spatial
Spatial.equals(Geometry, Geometry)	TRUE if the two geometries are spatially equal.	OGC SF	!	✔	! [2]	✔
Spatial.disjoint(Geometry, Geometry)	TRUE if the two geometries are spatially disjoint.	OGC SF	!	✔	! [2]	✔
Spatial.touches(Geometry, Geometry)	TRUE if the first Geometry spatially touches the other Geometry.	OGC SF	!	✔	! [2]	✔
Spatial.within(Geometry, Geometry)	TRUE if first Geometry is completely contained in second Geometry.	OGC SF	!	✔	! [2]	✔

Method	Description	Specification	Result [1]	PostGIS	MySQL	Oracle Spatial
Spatial.overlaps(Geometry, Geometry)	TRUE if first Geometries is spatially overlapping the other Geometry.	OGC SF	!	✔	! [2]	✔
Spatial.crosses(Geometry, Geometry)	TRUE if first Geometry crosses the other Geometry.	OGC SF	!	✔	✖ [3]	✔
Spatial.intersects(Geometry, Geometry)	TRUE if first Geometry spatially intersects the other Geometry.	OGC SF	!	✔	! [2]	✔
Spatial.contains(Geometry, Geometry)	TRUE if second Geometry is completely contained in first Geometry.	OGC SF	!	✔	! [2]	✔
Spatial.relate(Geometry, Geometry, String)	TRUE if the spatial relationship specified by the patternMatrix holds.	OGC SF	!	✔	✔	✔

[1] Oracle does not allow boolean expressions in the SELECT-list. [2] MySQL does not implement these functions according to the specification. They return the same result as the corresponding MBR-based functions.

Function on Distance Relationships (OGC SF 3.2.20)

Method	Description	Specification	Result [1]	PostGIS	MySQL	Oracle Spatial
Spatial.distance(Geometry, Geometry)	Returns the distance between the two geometries.	OGC SF	✔	✔	✔ [1]	✔

[1] MariaDB 5.3.3+ implements this.

Functions that implement Spatial Operators (OGC SF 3.2.21)

Method	Description	Specification	Result [1]	PostGIS	MySQL	Oracle Spatial
Spatial.intersection(Geometry, Geometry)	Returns a Geometry that is the set intersection of the two geometries.	OGC SF	✔	✔	✖	✔

Method	Description	Specification	Result [1]	PostGIS	MySQL	Oracle Spatial
Spatial.difference(Geometry, Geometry)	Returns a Geometry that is the closure of the set difference of the two geometries.	OGC SF	✔	✔	✖	✔
Spatial.union(Geometry, Geometry)	Returns a Geometry that is the set union of the two geometries.	OGC SF	✔	✔	✖	✔
Spatial.symDifference(Geometry, Geometry)	Returns a Geometry that is the closure of the set symmetric difference of the two geometries.	OGC SF	✔	✔	✖	✔
Spatial.buffer(Geometry, Double)	Returns as Geometry defined by buffering a distance around the Geometry.	OGC SF	✔	✔	✖	✔
Spatial.convexHull(Geometry)	Returns a Geometry that is the convex hull of the Geometry.	OGC SF	✔	✔	✖	✔

[1] These functions are currently not implemented in MySQL. They may appear in future releases.

Test whether the bounding box of one geometry intersects the bounding box of another

These functions are only available to specific RDBMS.

Method	Description	Result [1]	PostGIS	MySQL	Oracle Spatial
Spatial.bboxTest(Geometry, Geometry)	Returns TRUE if if the bounding box of the first Geometry overlaps second Geometry's bounding box	! [1]	✔	✔	✔

[1] Oracle does not allow boolean expressions in the SELECT-list.

PostGIS Spatial Operators

These functions are only supported on PostGIS.

Method	Description	Result
PostGIS.bboxOverlapsLeft(Geometry, Geometry)	The PostGIS &< operator returns TRUE if the bounding box of the first Geometry overlaps or is to the left of second Geometry's bounding box	✔
PostGIS.bboxOverlapsRight(Geometry, Geometry)	The PostGIS &< operator returns TRUE if the bounding box of the first Geometry overlaps or is to the right of second Geometry's bounding box	✔
PostGIS.bboxLeft(Geometry, Geometry)	The PostGIS << operator returns TRUE if the bounding box of the first Geometry overlaps or is strictly to the left of second Geometry's bounding box	✔
PostGIS.bboxRight(Geometry, Geometry)	The PostGIS << operator returns TRUE if the bounding box of the first Geometry overlaps or is strictly to the right of second Geometry's bounding box	✔
PostGIS.bboxOverlapsBelow(Geometry, Geometry)	The PostGIS &<@ operator returns TRUE if the bounding box of the first Geometry overlaps or is below second Geometry's bounding box	✔
PostGIS.bboxOverlapsAbove(Geometry, Geometry)	The PostGIS |&< operator returns TRUE if the bounding box of the first Geometry overlaps or is above second Geometry's bounding box	✔
PostGIS.bboxBelow(Geometry, Geometry)	The PostGIS <<| operator returns TRUE if the bounding box of the first Geometry is strictly below second Geometry's bounding box	✔
PostGIS.bboxAbove(Geometry, Geometry)	The PostGIS |<< operator returns TRUE if the bounding box of the first Geometry is strictly above second Geometry's bounding box	✔
PostGIS.sameAs(Geometry, Geometry)	The PostGIS ~= operator returns TRUE if the two geometries are vertex-by-vertex equal.	✔
PostGIS.bboxWithin(Geometry, Geometry)	The PostGIS @ operator returns TRUE if the bounding box of the first Geometry overlaps or is completely contained by second Geometry's bounding box	✔
PostGIS.bboxContains(Geometry, Geometry)	The PostGIS ~ operator returns TRUE if the bounding box of the first Geometry completely contains second Geometry's bounding box	✔

MySQL specific Functions for Testing Spatial Relationships between Minimal Bounding Boxes

These functions are only supported on MySQL

Method	Desription	Result
MySQL.mbrEqual(Geometry, Geometry)		✔
MySQL.mbrDisjoint(Geometry, Geometry)		✔
MySQL.mbrIntersects(Geometry, Geometry)		✔
MySQL.mbrTouches(Geometry, Geometry)		✔
MySQL.mbrWithin(Geometry, Geometry)		✔
MySQL.mbrContains(Geometry, Geometry)		✔
MySQL.mbrOverlaps(Geometry, Geometry)		✔

Oracle specific Functions for Constructing SDO_GEOMETRY types

These functions are only supported on Oracle Geospatial.

Method	Desription
Oracle.sdo_geometry(Integer gtype, Integer srid, SDO_POINT point, SDO_ELEM_INFO_ARRAY elem_info, SDO_ORDINATE_ARRAY ordinates)	Creates a SDO_GEOMETRY geometry from the passed geometry type, srid, point, element infos and ordinates.
Oracle.sdo_point_type(Double x, Double y, Double z)	Creates a SDO_POINT geometry from the passed ordinates.
Oracle.sdo_elem_info_array(String numbers)	Creates a SDO_ELEM_INFO_ARRAY from the passed comma-separeted integers.
Oracle.sdo_ordinate_array(String ordinates)	Creates a SDO_ORDINATE_ARRAY from the passed comma-separeted doubles.

Examples

The following sections provide some examples of what can be done using spatial methods in JPQL queries. In the examples we use a class from the test suite. Here's the source code for reference:

```
package mydomain.samples.pggeometry;

import org.postgis.LineString;

public class SampleLineString
{
    private long id;
    private String name;
    private LineString geom;

    public SampleLineString(long id, String name, LineString lineString)
    {
        this.id = id;
        this.name = name;
        this.geom = lineString;
    }

    public long getId()
    {
        return id;
    }
    ....
}
```

```
<entity-mappings>
    <package>mydomain.samples.pggeometry</package>

    <entity class="mydomain.samples.pggeometry.SampleLineString">
        <extension vendor-name="datanucleus" key="spatial-dimension" value="2"/>
        <extension vendor-name="datanucleus" key="spatial-srid" value="4326"/>
        <attributes>
            <id name="id"/>
            <basic name="name"/>
            <basic name="geom">
                <extension vendor-name="datanucleus" key="mapping" value="no-
userdata"/> [2]
            </basic>
        </attributes>
    </entity>
</entity-mappings>
```

Example 1 - Spatial Function in the Filter of a Query

This example shows how to use spatial functions in the filter of a query. The query returns a list of *SampleLineString(s)* whose line string has a length less than the given limit.

```
Query q = em.createQuery("SELECT s FROM SampleLineString s WHERE s.geom IS NOT NULL
AND Spatial.length(s.geom) < :limit");
q.setParameter("limit", new Double(100.0));
List list = q.getResultList();
```

Example 2 - Spatial Function in the Result Part of a Query

This time we use a spatial function in the result part of a query. The query returns the length of the line string from the selected *SampleLineString*

```
q = em.createQuery("SELECT Spatial.pointN(s.geom, 2) FROM SampleLineString s WHERE
s.id == :id");
q.setParameter("id", new Long(1001));
Geometry point = q.getSingleResult();
```

Example 3 - Nested Functions

You may want to use nested functions in your query. This example shows how to do that. The query returns a list of *SampleLineString(s)*, whose end point spatially equals a given point.

```
Point point = new Point("SRID=4326;POINT(110 45)");
Query q = em.createQuery("SELECT s FROM SampleLineString s WHERE s.geom IS NOT NULL
AND Spatial.equals(Spatial.endPoint(s.geom), :point)");
q.setParameter("point", point);
List list = q.getResultList();
```

Criteria

In JPA there is a query API referred to as "criteria", that broadly mirrors the JPQL query syntax. This is really an API allowing the construction of queries expression by expression, and optionally making it type-safe. It provides two ways of specifying a field/property. The first way is using Strings, and the second using a Static MetaModel. The advantage of the Static MetaModel is that it means that your queries are refactorable if you rename a field. Each example will be expressed in both ways where appropriate so you can see the difference.

Creating a Criteria query

To use the JPA Criteria API, firstly you need to create a *CriteriaQuery* object for the candidate in question, and set the candidate, its alias, and the result to be of the candidate type

```
CriteriaBuilder cb = emf.getCriteriaBuilder();
CriteriaQuery<Person> crit = cb.createQuery(Person.class);
Root<Person> candidateRoot = crit.from(Person.class);
candidateRoot.alias("p");

crit.select(candidateRoot);
```

So what we have there equates to

```
SELECT p FROM mydomain.Person p
```

For a complete list of all methods available on CriteriaBuilder, refer to Javadoc

For a complete list of all methods available on CriteriaQuery, refer to Javadoc

JPQL equivalent of the Criteria query

Extension

If you ever want to know what is the equivalent JPQL string-based query for your Criteria, just print out *criteriaQuery.toString()*. This is **not** part of the JPA spec, but something that we feel is very useful so is provided as a DataNucleus vendor extension. So, for example, the criteria query above would result in the following from *crit.toString()*

```
SELECT p FROM mydomain.Person p
```

Criteria API : Result clause

The basic Criteria query above is fine, but you may want to define a result other than the candidate. To do this we need to use the Criteria API.

```
Path nameField = candidateRoot.get("name");
crit.select(nameField);
```

which equates to

```
SELECT p.name
```

Note that here we accessed a field by its name (as a String). We could easily have accessed it via the Criteria MetaModel too.

Criteria API : From clause joins

The basic Criteria query above is fine, but you may want to define some explicit joins. To do this we need to use the Criteria API.

```
// String-based:
ManagedType personType = emf.getMetamodel().type(Person.class);
Attribute addressAttr = personType.getAttribute("address");
Join addressJoin = candidateRoot.join((SingularAttribute)addressAttr);
addressJoin.alias("a");

// MetaModel-based:
Join<Person, Address> addressJoin = candidateRoot.join(Person_.address);
addressJoin.alias("a");
```

which equates to

```
FROM mydomain.Person p JOIN p.address a
```

Criteria API : Filter

The basic Criteria query above is fine, but in the majority of cases we want to define a filter. To do this we need to use the Criteria API.

```
// String-based:
Predicate nameEquals = cb.equal(candidateRoot.get("name"), "First");
crit.where(nameEquals);

// MetaModel-based:
Predicate nameEquals = cb.equal(candidateRoot.get(Person_.name), "First");
crit.where(nameEquals);
```

You can also invoke methods, so a slight variation on this clause would be

```
// String-based:
Predicate nameUpperEquals = cb.equal(cb.upper(candidateRoot.get("name")), "FIRST");
crit.where(nameUpperEquals);

// MetaModel-based:
Predicate nameUpperEquals = cb.equal(cb.upper(candidateRoot.get(Person_.name)),
"FIRST");
crit.where(nameUpperEquals);
```

which equates to

```
WHERE (UPPER(p.name) = 'FIRST')
```

You can combine predicates into AND/OR structures like this

```
// String-based:
Predicate nameEquals = cb.equal(candidateRoot.get("name"), "First");
Predicate ageEquals = cb.equal(candidateRoot.get("age"), 18);
crit.where(cb.and(nameEquals, ageEquals));

// MetaModel-based:
Predicate nameEquals = cb.equal(candidateRoot.get(Person_.name), "First");
Predicate ageEquals = cb.equal(candidateRoot.get(Person_.age), 18);
crit.where(b.and(nameEquals, ageEquals));
```

```
// String-based:
Predicate nameEquals = cb.equal(candidateRoot.get("name"), "First");
Predicate ageEquals = cb.equal(candidateRoot.get("age"), 18);
crit.where(cb.or(nameEquals, ageEquals));

// MetaModel-based:
Predicate nameEquals = cb.equal(candidateRoot.get(Person_.name), "First");
Predicate ageEquals = cb.equal(candidateRoot.get(Person_.age), 18);
crit.where(b.or(nameEquals, ageEquals));
```

Criteria API : Ordering

The basic Criteria query above is fine, but in many cases we want to define ordering. To do this we need to use the Criteria API.

```
// String-based:
Order orderName = cb.desc(candidateRoot.get("name"));
crit.orderBy(orderName);

// MetaModel-based:
Order orderName = cb.desc(candidateRoot.get(Person_.name));
crit.orderBy(orderName);
```

which equates to

```
ORDER BY p.name DESC
```

DataNucleus provides an extension to the JPA Criteria API in its *javax.persistence-2.2.jar* where you have additional methods to specify where NULL values are placed in the ordering. Like this

```
Order orderName = cb.desc(candidateRoot.get("name"));
orderName.nullsFirst();
```

which will put NULL values of that field before other values. Similarly there is a method `nullsLast`.

Criteria API : Parameters

Another common thing we would want to do is specify input parameters. To do this we need to use the Criteria API. Let's take an example of a filter with parameters.

```
// String-based:
ParameterExpression param1 = cb.parameter(String.class, "myParam1");
Predicate nameEquals = cb.equal(candidateRoot.get("name"), param1);
crit.where(nameEquals);

// MetaModel-based:
ParameterExpression param1 = cb.parameter(String.class, "myParam1");
Predicate nameEquals = cb.equal(candidateRoot.get(Person_.name), param1);
crit.where(nameEquals);
```

which equates to

```
WHERE (p.name = :myParam)
```

Don't forget to set the value of the parameters before executing the query!

Criteria API : Subqueries

You can also make use of subqueries with Criteria.

In this example we are going to search for all *Employee(s)* where the salary is below the average of all *Employees*. In JPQL this would be written as

```
SELECT e FROM Employee e WHERE (e.salary < SELECT AVG(e2.salary) FROM Employee e2")
```

With Criteria we do it like this. Firstly we create the outer query, then create the subquery, and then place the subquery in the outer query.

```
CriteriaQuery<Employee> crit = cb.createQuery(Employee.class);
Root<Employee> candidate = crit.from(Employee.class);
candidate.alias("e");
crit.select(candidate);

// Create subquery for the average salary of all Employees
Subquery<Double> subCrit = crit.subquery(Double.class);
Root<Employee> subCandidate = subCrit.from(Employee.class);
subCandidate.alias("e2");
Path e2SalaryField = subCandidate.get("salary");
Subquery<Double> avgSalary = subCrit.select(cb.avg(e2SalaryField));

// Add WHERE clause to outer query, linking to subquery
Path eSalaryField = candidate.get("salary");
Predicate lessThanAvgSalary = cb.lessThan(eSalaryField, avgSalary);
crit.where(lessThanAvgSalary);
```

Criteria API : IN operator

You can make use of the IN operator with Criteria, like this

```
List<String> nameOptions = new ArrayList<String>();
nameOptions.add("Fred");
nameOptions.add("George");

Path nameField = candidateRoot.get("name");
Predicate nameIn = nameField.in(nameOptions);
```

so this generates the equivalent of this JPQL

```
name IN ('Fred', 'George')
```

An alternative way of doing this is via the CriteriaBuilder

```
In nameIn = cb.in(candidateRoot.get("name"));
nameIn.value("Fred");
nameIn.value("George");

crit.where(nameIn);
```

Criteria API : Result as Tuple

You sometimes need to define a result for a query. You can define a result class just like with normal JPQL, but a special case is where you don't have a particular result class and want to use the *built-in* JPA standard **Tuple** class.

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

Executing a Criteria query

Ok, so we've seen how to generate a Criteria query. So how can we execute it ? This is simple; convert it into a standard JPA query, set any parameter values and execute it.

```
Query query = em.createQuery(crit);
List<Person> results = query.getResultList();
```

Criteria API : UPDATE query

So the previous examples concentrated on SELECT queries. Let's now do an UPDATE

```
// String-based:
CriteriaUpdate<Person> crit = qb.createCriteriaUpdate(Person.class);
Root<Person> candidate = crit.from(Person.class);
candidate.alias("p");
crit.set(candidate.get("firstName"), "Freddie");
Predicate teamName = qb.equal(candidate.get("firstName"), "Fred");
crit.where(teamName);
Query q = em.createQuery(crit);
int num = q.executeUpdate();

// MetaModel-based:
CriteriaUpdate<Person> crit = qb.createCriteriaUpdate(Person.class);
Root<Person> candidate = crit.from(Person.class);
candidate.alias("p");
crit.set(candidate.get(Person_.firstName), "Freddie");
Predicate teamName = qb.equal(candidate.get(Person.firstName), "Fred");
crit.where(teamName);
Query q = em.createQuery(crit);
int num = q.executeUpdate();
```

which equates to

```
UPDATE Person p SET p.firstName = 'Freddie' WHERE p.firstName = 'Fred'
```

Criteria API : DELETE query

So the previous examples concentrated on SELECT queries. Let's now do a DELETE

```
// String-based:
CriteriaDelete<Person> crit = qb.createCriteriaDelete(Person.class);
Root<Person> candidate = crit.from(Person.class);
candidate.alias("p");
Predicate teamName = qb.equal(candidate.get("firstName"), "Fred");
crit.where(teamName);
Query q = em.createQuery(crit);
int num = q.executeUpdate();

// MetaModel-based:
CriteriaDelete<Person> crit = qb.createCriteriaDelete(Person.class);
Root<Person> candidate = crit.from(Person.class);
candidate.alias("p");
Predicate teamName = qb.equal(candidate.get(Person.firstName), "Fred");
crit.where(teamName);
Query q = em.createQuery(crit);
int num = q.executeUpdate();
```

which equates to

```
DELETE FROM Person p WHERE p.firstName = 'Fred'
```

Static MetaModel

As we mentioned at the start of this section, there is a Static MetaModel allowing refactorability. In JPA the MetaModel is a *static metamodel* of generated classes that mirror the applications entities and have persistable fields marked as *public* and *static* so that they can be accessed when generating the queries. In the examples above you saw reference to a class with name with suffix "_". This is a (static) metamodel class. It is defined below.

The JPA spec contains the following description of the static metamodel.

For every managed class in the persistence unit, a corresponding metamodel class is produced as follows:

- For each managed class X in package p, a metamodel class X_ in package p is created.
- The name of the metamodel class is derived from the name of the managed class by appending "_" to the name of the managed class.
- The metamodel class X_ must be annotated with the javax.persistence.StaticMetamodel annotation
- If class X extends another class S, where S is the most derived managed class (i.e., entity or mapped superclass) extended by X, then class X_ must extend class S_, where S_ is the metamodel class created for S.
- For every persistent non-collection-valued attribute y declared by class X, where the type of y is Y, the metamodel class must contain a declaration as follows:

```
public static volatile SingularAttribute<X, Y> y;
```

- For every persistent collection-valued attribute z declared by class X, where the element type of z is Z, the metamodel class must contain a declaration as follows:
  - if the collection type of z is java.util.Collection, then

```
public static volatile CollectionAttribute<X, Z> z;
```

- if the collection type of z is java.util.Set, then

```
public static volatile SetAttribute<X, Z> z;
```

- if the collection type of z is java.util.List, then

```
public static volatile ListAttribute<X, Z> z;
```

- if the collection type of z is java.util.Map, then

```
public static volatile MapAttribute<X, K, Z> z;
```

where K is the type of the key of the map in class X

Let's take an example, for the following class

```
package mydomain.metamodel;

import java.util.*;
import javax.persistence.*;

@Entity
public class Person
{
    @Id
    long id;

    String name;

    @OneToMany
    List<Address> addresses;
}
```

the static metamodel class (generated by `datanucleus-jpa-query.jar`) will be

```
package mydomain.metamodel;

import javax.persistence.metamodel.*;

@StaticMetamodel(Person.class)
public class Person_
{
    public static volatile SingularAttribute<Person, Long> id;
    public static volatile SingularAttribute<Person, String> name;
    public static volatile ListAttribute<Person, Address> addresses;
}
```

So how do we generate this metamodel definition for our query classes? DataNucleus provides an *annotation processor* in `datanucleus-jpa-query.jar` that can be used when compiling your model classes to generate the static metamodel classes. What this does is when the compile is invoked, all classes that have persistence annotations will be passed to the annotation processor and a Java file generated for its metamodel. Then all classes (original + metamodel) are compiled.

Using Maven

To enable this in Maven you would need the above jar as well as `javax.persistence.jar` to be in the CLASSPATH at compile This creates the "static metamodel" classes under *target/generated-sources/annotations/*. You can change this location using the configuration property **generatedSourcesDirectory** of the *maven-compiler-plugin*.

Using Eclipse

To enable this in Eclipse you would need to do the following

- Go to *Java Compiler* and make sure the compiler compliance level is 1.8 or above (needed for DN 5+ anyway)
- Go to *Java Compiler* → *Annotation Processing* and enable the project specific settings and enable annotation processing
- Go to *Java Compiler* → *Annotation Processing* → *Factory Path*, enable the project specific settings and then add the following jars to the list: `datanucleus-jpa-query.jar`, `javax.persistence.jar`

This creates the "static metamodel" classes under *target/generated-sources/annotations/*. You can change this location on the *Java Compiler* → *Annotation Processing* page.

Native Queries

The JPA specification defines its interpretation of native queries, for selecting objects from the datastore. To provide a simple example for RDBMS (i.e using SQL), this is what you would do

```
Query q = em.createNativeQuery("SELECT p.id, o.firstName, o.lastName FROM Person p,
Job j WHERE (p.job = j.id) AND j.name = 'Cleaner'");
List results = (List)q.getResultsList();
```

This finds all "Person" objects that do the job of "Cleaner". The syntax chosen has to be runnable on the RDBMS that you are using (and since SQL is anything but "standard" you will likely have to change your query when moving to another datastore).

Input Parameters

In queries it is convenient to pass in parameters so we don't have to define the same query for different values. Here's an example

```
// Numbered Parameters :
Query q = em.createQuery("SELECT p.id FROM Person p WHERE p.lastName = ?1 AND
p.firstName = ?2");
q.setParameter(1, theSurname).setParameter(2, theForename);
```

So we have parameters that are prefixed by **?** (question mark) and are numbered starting at 1. We then use the numbered position when calling *Query.setParameter()*. This is known as *numbered* parameters. With JPA native queries we can't use named parameters officially.

DataNucleus also actually supports use of *named* parameters where you assign names just like in JPQL. This is not defined by the JPA specification so dont expect other JPA implementations to support it. Let's take the previous example and rewrite it using *named* parameters, like this

```
// Named Parameters :
Query q = em.createQuery("SELECT p.id FROM Person p WHERE p.lastName = :firstParam AND
p.firstName = :otherParam");
q.setParameter("firstParam", theSurname).setParameter("otherParam", theForename);
```

Range of Results

With SQL you can select the range of results to be returned. For example if you have a web page and you are paginating the results of some search, you may want to get the results from a query in blocks of 20 say, with results 0 to 19 on the first page, then 20 to 39, etc. You can facilitate this as follows

```
Query q = em.createNativeQuery("SELECT p.id FROM Person p WHERE p.age > 20");
q.setFirstResult(0).setMaxResults(20);
```

So with this query we get results 0 to 19 inclusive.

SQL Syntax Checks

When a native query is a SELECT, and is returning instances of an Entity, then it is required to return the columns for the PK, version and discriminator (if applicable). DataNucleus provides some checks that can be performed to ensure that these are selected. You can turn this checking off by setting the persistence property **datanucleus.sql.syntaxChecks** to *false*. Similarly you can turn them off on a query-by-query basis by setting the query hint **datanucleus.sql.syntaxChecks** to *false*.

Query Execution

There are two ways to execute a native query. When you know it will return 0 or 1 results you call

```
Object result = query.getSingleResult();
```

If however you know that the query will return multiple results, or you just don't know then you would call

```
List results = query.getResultList();
```

SQL Result Definition

By default, if you simply execute a native query and don't specify the result mapping, then when you execute *getResultList()* each row of the results will be an Object array. You can however define how the results are mapped to some result class for example. Let's give some examples of what you can do. If we have the following entities

```
@Entity
@Table(name="LOGIN")
public class Login
{
    @Id
    private long id;

    private String userName;
    private String password;

    public Login(String user, String pwd)
    {
        ...
    }
}

@Entity
@Table(name="LOGINACCOUNT")
public class LoginAccount
{
    @Id
    private long id;

    private String firstName;
    private String lastName;

    @OneToOne(cascade={CascadeType.MERGE, CascadeType.PERSIST}, orphanRemoval=true)
    @JoinColumn(name="LOGIN_ID")
    private Login login;

    public LoginAccount(long id, String firstName, String lastName)
    {
        ...
    }
}
```

The first thing to do is to select both LOGIN and LOGINACCOUNT columns in a single call, and return instances of the 2 entities. So we define the following in the *LoginAccount* class

```
@SqlResultSetMappings({
    @SqlResultSetMapping(name="LOGIN_PLUS_ACCOUNT",
        entities={@EntityResult(entityClass=LoginAccount.class), @EntityResult
(entityClass=Login.class)})
```

and we now execute the native query as

```
List<Object[]> result = em.createNativeQuery("SELECT P.ID, P.FIRSTNAME, P.LASTNAME,
P.LOGIN_ID, L.ID, L.USERNAME, L.PASSWORD " +
    "FROM JPA_AN_LOGINACCOUNT P, JPA_AN_LOGIN L", "AN_LOGIN_PLUS_ACCOUNT"
).getResultList();
Iterator iter = result.iterator();
while (iter.hasNext())
{
    Object[] row = iter.next();
    LoginAccount acct = (LoginAccount)obj[0];
    Login login = (Login)obj[1];
    ...
}
```

Next thing to try is the same as above, returning 2 entities for a row, but here we explicitly define the mapping of SQL column to the constructor parameter.

```
@SqlResultSetMapping(name="AN_LOGIN_PLUS_ACCOUNT_ALIAS", entities={
            @EntityResult(entityClass=LoginAccount.class, fields={@FieldResult(name
="id", column="THISID"), @FieldResult(name="firstName", column="FN")}),
            @EntityResult(entityClass=Login.class, fields={@FieldResult(name="id",
column="IDLOGIN"), @FieldResult(name="userName", column="UN")})
        })
```

and we now execute the native query as

```
List<Object[]> result = em.createNativeQuery("SELECT P.ID AS THISID, P.FIRSTNAME AS
FN, P.LASTNAME, P.LOGIN_ID, " +
    "L.ID AS IDLOGIN, L.USERNAME AS UN, L.PASSWORD FROM JPA_AN_LOGINACCOUNT P,
JPA_AN_LOGIN L", "AN_LOGIN_PLUS_ACCOUNT_ALIAS").getResultList();
Iterator iter = result.iterator();
while (iter.hasNext())
{
    Object[] row = iter.next();
    LoginAccount acct = (LoginAccount)obj[0];
    Login login = (Login)obj[1];
    ...
}
```

For our final example we will return each row as a non-entity class, defining how the columns map to the constructor for the result class.

```
@SqlResultSetMapping(name="AN_LOGIN_PLUS_ACCOUNT_CONSTRUCTOR", classes={
           @ConstructorResult(targetClass=LoginAccountComplete.class,
               columns={@ColumnResult(name="FN"), @ColumnResult(name="LN"),
@ColumnResult(name="USER"), @ColumnResult(name="PWD")}),
        })
```

with non-entity result class defined as

```
public class LoginAccountComplete
{
    String firstName;
    String lastName;
    String userName;
    String password;

    public LoginAccountComplete(String firstName, String lastName, String userName,
String password)
    {
        ...
    }
    ...
}
```

and we execute the query like this

```
List result = em.createNativeQuery("SELECT P.FIRSTNAME AS FN, P.LASTNAME AS LN,
L.USERNAME AS USER, L.PASSWORD AS PWD FROM " +
    "JPA_AN_LOGINACCOUNT P, JPA_AN_LOGIN L","AN_LOGIN_PLUS_ACCOUNT_CONSTRUCTOR"
).getResultList();
Iterator iter = result.iterator();
while (iter.hasNext())
{
    LoginAccountComplete acctCmp = (LoginAccountComplete)iter.next();
    ...
}
```

Named Native Query

With the JPA API you can either define a query at runtime, or define it in the MetaData/annotations for a class and refer to it at runtime using a symbolic name. This second option means that the method of invoking the query at runtime is much simplified. To demonstrate the process, lets say we have a class called *Product* (something to sell in a store). We define the JPA Meta-Data for the class in the normal way, but we also have some query that we know we will require, so we define the following in the Meta-Data.

```
<entity class="Product">
    ...
    <named-native-query name="PriceBelowValue"><![CDATA[
    SELECT NAME FROM PRODUCT WHERE PRICE < ?1
    ]]></named-native-query>
</entity>
```

or using annotations

```
@Entity
@NamedNativeQuery(name="PriceBelowValue", query="SELECT NAME FROM PRODUCT WHERE PRICE
< ?1")
public class Product {...}
```

So here we have a native query that will return the names of all Products that have a price less than a specified value. This leaves us the flexibility to specify the value at runtime. So here we run our named native query, asking for the names of all Products with price below 20 euros.

```
Query query = em.createNamedQuery("PriceBelowValue");
List results = query.setParameter(1, new Double(20.0)).getResultList();
```

Cassandra Native (CQL) Queries

If you choose to use Cassandra CQL Queries then these are not portable to any other datastore. Use JPQL for portability

Cassandra provides the CQL query language. To take a simple example using it with the JPA API and a Cassandra datastore

```
// Find all employees
Query q = em.createNativeQuery("SELECT * FROM schema1.Employee", Employee.class);
List<Employee> results = (List)q.getResultList();
```

Note that the last argument to *createNativeQuery* is optional and you would get *List<Object[]>* returned otherwise.

Stored Procedures

applicable to RDBMS.

The JPA specification supports calling stored procedures through its API. It allows some flexibility in the type of stored procedure being used, supporting IN/OUT/INOUT parameters as well as result sets being returned. Obviously if a particular RDBMS does not support stored procedures then this functionality will not apply.

You start off by creating a stored procedure query, like this, referencing the stored procedure name in the datastore.

```
StoredProcedureQuery spq = em.createStoredProcedureQuery("PERSON_SP_1");
```

You should familiarise yourself with the StoredProcedureQuery Javadoc API.

If we have any parameters in this stored procedure we need to register them, for example

```
spq.registerStoredProcedureParameter("PARAM1", String.class, ParameterMode.IN);
spq.registerStoredProcedureParameter("PARAM2", Integer.class, ParameterMode.OUT);
```

If you have any result class, or result set mapping then you can specify those in the *createStoredProcedureQuery* call. Now we are ready to execute the query and access the results.

Simple execution, returning a result set

A common form of stored procedure will simply return a single result set. You execute such a procedure as follows

```
List results = spq.getResultList();
```

or if expecting a single result, then

```
Object result = spq.getSingleResult();
```

Simple execution, returning output parameters

A common form of stored procedure will simply return output parameter(s). You execute such a procedure as follows

```
spq.execute();
Object paramVal = spq.getOutputParameterValue("PARAM2");
```

or you can also access the output parameters via position (if specified by position).

Generalised execution, for multiple result sets

A more complicated, yet general, form of execution of the stored procedure is as follows

```
boolean isResultSet = spq.execute(); // returns true when we have a result set from
the proc
List results1 = spq.getResultList(); // get the first result set
if (spq.hasMoreResults())
{
    List results2 = spq.getResultList(); // get the second result set
}
```

So the user can get hold of multiple result sets returned by their stored procedure.

Named Stored Procedure Queries

Just as with normal queries, you can also register a stored procedure query at development time and then access it via name from the EntityManager. So we define one like this (not important on which class it is defined)

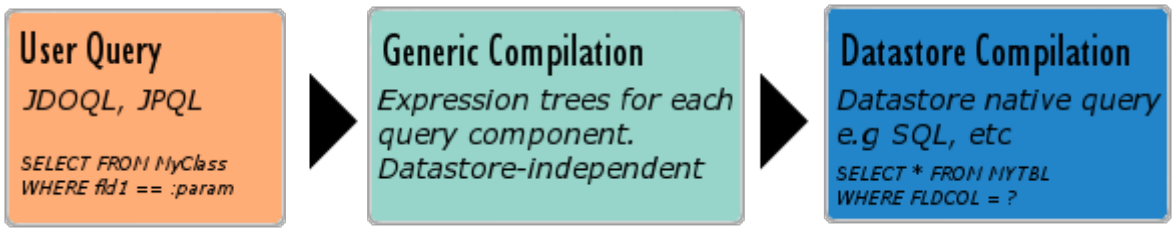
```
@NamedStoredProcedureQuery(name="myTestProc", procedureName="MY_TEST_SP_1",
    parameters={@StoredProcedureParameter(name="PARAM1", type=String.class, mode
=ParameterMode.IN})

@Entity
public class MyClass {...}
```

and then create the query from the EntityManager

```
StoredProcedureQuery spq = em.createNamedStoredProcedureQuery("myTestProc");
```

Query Cache

Extension



JPA doesn't currently define a mechanism for caching of queries. DataNucleus provides 3 levels of caching

- Generic Compilation : when a query is compiled it is initially compiled *generically* into expression trees. This generic compilation is independent of the datastore in use, so can be used for other datastores. This can be cached.
- Datastore Compilation : after a query is compiled into expression trees (above) it is then converted into the native language of the datastore in use. For example with RDBMS, it is converted into SQL. This can be cached
- Results : when a query is run and returns objects of the candidate type, you can cache the identities of the result objects.

Generic Query Compilation Cache

This cache is by default set to *soft*, meaning that the generic query compilation is cached using soft references. This is set using the persistence property **datanucleus.cache.queryCompilation.type**. You can also set it to *strong* meaning that strong references are used, or *weak* meaning that weak references are used.

You can turn caching on/off (default = on) on a query-by-query basis by specifying the query extension **datanucleus.query.compilation.cached** as true/false.

Datastore Query Compilation Cache

This cache is by default set to *soft*, meaning that the datastore query compilation is cached using soft references. This is set using the persistence property **datanucleus.cache.queryCompilationDatastore.type**. You can also set it to *strong* meaning that strong references are used, or *weak* meaning that weak references are used.

You can turn caching on/off (default = on) on a query-by-query basis by specifying the query extension **datanucleus.query.compilation.cached** as true/false. As a finer degree of control, where cached results are used, you can omit the validation of object existence in the datastore by setting the query extension **datanucleus.query.resultCache.validateObjects**.

Query Results Cache

This cache is by default set to *soft*, meaning that the datastore query results are cached using soft references. This is set using the persistence property **datanucleus.cache.queryResults.type**. You can also set it to *strong* meaning that strong references are used, or *weak* meaning that weak references are used.

You can specify persistence property **datanucleus.cache.queryResults.cacheName** to define the name of the cache used for the query results cache.

You can specify persistence property **datanucleus.cache.queryResults.expireMillis** to specify the expiry of caching of results, for caches that support it.

You can specify persistence property **datanucleus.cache.queryResults.maxSize** to define the maximum number of queries that have their results cached, for caches that support it.

You can turn caching on/off (default = off) on a query-by-query basis by specifying the query extension **datanucleus.query.results.cached** as true/false.

Obviously with a cache of query results, you don't necessarily want to retain this cached over a long period. In this situation you can evict results from the cache like this.

```
import org.datanucleus.api.jpa.JPAQueryCache;
import org.datanucleus.api.jpa.JPAEntityManagerFactory;

...
JPAQueryCache cache = ((JPAEntityManagerFactory)emf).getQueryCache();

cache.evict(query);
```

which evicts the results of the specific query. The JPAQueryCache Javadoc has more options available should you need them.