JDO Query Guide (v5.2)

Table of Contents

Query API . . . 2
- Creating a query . . . 2
- Closing a query . . . 3
- Named Query . . . 3
- Query Extensions . . . 5
- Setting query parameters . . . 6
- Compiling a query . . . 6
- Executing a query . . . 6
- Controlling the execution : FetchPlan . . . 7
- ignoreCache(), setIgnoreCache() . . . 9
- Control over locking of fetched objects . . . 9
- Timeout on query execution for reads . . . 9
- Timeout on query execution for writes . . . 10
- Extension: Loading Large Result Sets at Commit() . . . 10
- Extension: Caching of Results . . . 10
- Extension: Size of Large Result Sets . . . 10
- Extension: Type of Result Set (RDBMS) . . . 11
- Extension: Result Set Control (RDBMS) . . . 11

JDOQL . . . 13
- JDOQL Single-String syntax . . . 13
- Candidate Class . . . 14
- Filter . . . 15
- Fields/Properties . . . 16
- Methods . . . 17
- Literals . . . 35
- Parameters . . . 35
- Variables . . . 37
- Imports . . . 38
- IF ELSE expressions . . . 39
- Operators . . . 39
- instanceof . . . 40
- casting . . . 40
- Subqueries . . . 40
- Result clause . . . 43
- Result Class . . . 44
- Grouping of Results . . . 45
- Ordering of Results . . . 46
- Range of Results . . . 46

JDOQL In-Memory queries . . . 47
Update/Delete queries . . . 48
Deletion by Query . . . 48
Bulk Delete . . . 48
Bulk Update . . . 48
JDOQL Strictness . . . 49
JDOQL : SQL Generation for RDBMS . . . 49
JDOQL Typed . . . 50
Preparation . . . 50
Query Classes . . . 51
Query API - Filtering . . . 52
Query API - Ordering . . . 53
Query API - Methods . . . 53
Query API - Results . . . 55
Query API - Parameters . . . 55
Query API - Variables . . . 56
Query API - If-Then-Else . . . 56
Query API - Subqueries . . . 57
Query API - Candidates . . . 57
SQL . . . 59
Setting candidate class . . . 59
Unique results . . . 60
Defining a result type . . . 60
SQL Syntax Checks . . . 61
Inserting/Updating/Deleting . . . 61
Parameters . . . 62
Example 1 - Using SQL aggregate functions, without candidate class . . . 62
Example 2 - Using SQL aggregate functions, with result class . . . 63
Example 3 - Retrieval using candidate class . . . 63
Example 4 - Using parameters, without candidate class . . . 64
Example 5 - Named Query . . . 64
Cassandra CQL . . . 66
JPQL . . . 67
Entity Name . . . 67
Fetched Fields . . . 67
Stored Procedures . . . 68
Using DataNucleus Stored Procedure API . . . 68
Using JDO SQL Query API to invoke stored procedures . . . 69
Query Cache . . . 70
Generic Query Compilation Cache . . . 70
Datastore Query Compilation Cache . . . 70

Query Results Cache 71

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. JDO specifies support for

- JDOQL : a string-based query language using Java syntax.
- Typed : following JDOQL syntax but providing an API supporting refactoring of classes and the queries they are used in.
- SQL : typically only for RDBMS
- JPQL : not explicitly part of the JDO spec, but provided by DataNucleus JDO.
- Stored Procedures : not explicitly part of the JDO spec, but provided by DataNucleus JDO as an option for RDBMS.

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 **JDOQL**. 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.

We recommend using JDOQL for queries wherever possible since it is object-based and datastore agnostic, giving you extra flexibility in the future. If not possible using JDOQL, only then use a language appropriate to the datastore in question

For some datastores additional query languages may be available specific to that datastore - please check the datastores documentation.

There are 2 categories of queries with JDO :-

- **Programmatic Query** where the query is defined using the JDO Query API.
- **Named Query** where the query is defined in MetaData and referred to by its name at runtime.

Query API

Let's now try to understand the Query Javadoc API in JDO. We firstly need to look at a typical Query.

Let's create a JDOQL string-based query to highlight its usage

```
Query q = pm.newQuery("SELECT FROM mydomain.Product p WHERE p.price <= :threshold
ORDER BY p.price ASC");
List results = q.execute(my_threshold);
```

In this Query, we implicitly select JDOQL by just passing in a query string to the method *PersistenceManager.newQuery(String)*, and the query is specified to return all objects of type *Product* (or subclasses) which have the price less than or equal to some threshold value and ordering the results by the price. 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 (JDOQL) string-based Query.

Creating a query

The principal ways of creating a query are

- Specifying the query language, and using a single-string form of the query

```
Query q = pm.newQuery("javax.jdo.query.JDOQL",
    "SELECT FROM mydomain.MyClass WHERE field2 < threshold PARAMETERS java.util.Date
threshold");
```

or alternatively

```
Query q = pm.newQuery("SQL", "SELECT * FROM MYTABLE WHERE COL1 == 25);
```

- A "named" query, (pre-)defined in metadata (refer to metadata docs).

```
Query<MyClass> q = pm.newNamedQuery(MyClass.class, "MyQuery1");
```

- JDOQL : Use the single-string form of the query

```
Query q = pm.newQuery("SELECT FROM mydomain.MyClass WHERE field2 < threshold
PARAMETERS java.util.Date threshold");
```

- JDOQL : Use the declarative API to define the query

```
Query<MyClass> q = pm.newQuery(MyClass.class);
q.setFilter("field2 < threshold");
q.declareParameters("java.util.Date threshold");
```

- JDOQL : Use the Typed Query API to define the query

```
JDOQLTypedQuery<MyClass> q = pm.newJDOQLTypedQuery(MyClass.class);
QMyClass cand = QMyClass.candidate();
List<Product> results =
q.filter(cand.field2.lt(q.doubleParameter("threshold"))).executeList();
```

Closing a query

When a query is executed it will have access to the results of that query. Each time it is executed (maybe with different parameters) it will have separate results. This can consume significant resources if the query returned a lot of records.

You close a query (and all query results) like this

```
q.close();
```

If you just wanted to close a specific query result you would call

```
q.close(queryResult);
```

where the *queryResult* is what you were returned from executing the query.

Named Query

With the JDO query 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 JDO 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.

```
<package name="mydomain">
    <class name="Product">
        ...
        <query name="SoldOut" language="javax.jdo.query.JDOQL"><![CDATA[
        SELECT FROM mydomain.Product WHERE status == "Sold Out"
        ]]></query>
    </class>
</package>
```

So we have a JDOQL query called "SoldOut" defined for the class *Product* that returns all Products (and subclasses) that have a *status* of "Sold Out". Out of interest, what we would then do in our application to execute this query woule be

```
Query<Product> q = pm.newNamedQuery(mydomain.Product.class,"SoldOut");
List<Product> results = q.executeList();
```

The above example was for the JDOQL object-based query language. We can do a similar thing using SQL, so we define the following in our MetaData for our *Product* class

```
<jdo>
    <package name="mydomain">
        <class name="Product">
            ...
            <query name="PriceBelowValue" language="javax.jdo.query.SQL"><![CDATA[
            SELECT NAME FROM PRODUCT WHERE PRICE < ?
            ]]></query>
        </class>
    </package>
</jdo>
```

So here we have an SQL 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 query, asking for the names of all Products with price below 20 euros.

```
Query<Product> q = pm.newNamedQuery(mydomain.Product.class, "PriceBelowValue");
q.setParameters(20.0);
List<Product> results = q.executeList();
```

All of the examples above have been specifed within the <class> element of the MetaData. You can, however, specify queries below <jdo> in which case the query is not scoped by a particular candidate class. In this case you must put your queries in any of the following MetaData files

```
/META-INF/package.jdo
/WEB-INF/package.jdo
/package.jdo
/META-INF/package-{mapping}.orm
/WEB-INF/package-{mapping}.orm
/package-{mapping}.orm
/META-INF/package.jdoquery
/WEB-INF/package.jdoquery
/package.jdoquery
```

Saving a Query as a Named Query

DataNucleus JDO also allows you to create a query, and then save it as a "named" query for later reuse. You do this as follows

```
Query q = pm.newQuery("SELECT FROM Product p WHERE ...");
q.saveAsNamedQuery("MyQuery");
```

and you can thereafter access the query via

```
Query q = pm.newNamedQuery(Product.class, "MyQuery");
```

Query Extensions

The JDO query API allows implementations to support "extensions" and provides a simple interface for enabling the use of such extensions on queries. An extension specifies additional information to the query mechanism about how to perform the query. Individual extensions will be explained later in this guide.

You set an extension like this

```
q.extension("extension_name", value);
```

```
Map exts = new HashMap();
exts.put("extension1", value1);
exts.put("extension2", value2);
q.extensions(exts);
```

With DataNucleus, all *extension names* will begin with "datanucleus.".

The Query API also has methods *setExtensions* and *addExtension* that are from the original version of the API, but function the same as these methods quoted.

Setting query parameters

Queries can be made flexible and reusable by defining parameters as part of the query, so that we can execute the same query with different sets of parameters and minimise resources.

```
// JDOQL Using named parameters
Query<Product> q = pm.newQuery(Product.class);
q.setFilter("this.name == :name && this.serialNo == :serial");

Map params = new HashMap();
params.put("name", "Walkman");
params.put("serial", "123021");
q.setNamedParameters(params);


// JDOQL Using numbered parameters
Query<Product> q = pm.newQuery(Product.class);
q.setFilter("this.name == ?1 && this.serialNo == ?2");

q.setParameters("Walkman", "123021");
```

Alternatively you can specify the query parameters in the *execute* method call.

Compiling a query

An intermediate step once you have your query defined, if you want to check its validity, is to *compile* it. You do this as follows

```
q.compile();
```

If the query is invalid, then a JDOException will be thrown.

Executing a query

So we have set up our query. We now execute it. We have various methods to do this, depending on what result we are expecting etc

```
// Simple execute
Object result = q.execute();

// Execute with 1 parameter passed in
Object result = q.execute(paramVal1);

// Execute with multiple parameters passed in
Object result = q.execute(paramVal1, paramVal2);

// Execute with an array of parameters passed in (positions match the query parameter
position)
Object result = q.executeWithArray(new Object[]{paramVal1, paramVal2});

// Execute with a map of parameters keyed by their name in the query
Object result = q.executeWithMap(paramMap);

// Execute knowing we want to receive a list of results
List results = q.executeList();

// Execute knowing there is 1 result row
Object result = q.executeUnique();

// Execute where we want a list of results and want each result row of a particular
type
List<ResultClass> results = q.executeResultList(ResultClass.class);

// Execute where we want a single result and want the result row of a particular type
ResultClass result = q.executeResultUnique(ResultClass.class);
```

Extension : Flush before query execution

When using optimistic transactions all updates to persistent objects are held until flush()/commit(). This means that executing a query may not take into account changes made during that transaction in some objects. DataNucleus allows an extension for calling flush() just before execution of queries so that all updates are taken into account. You could specify this as a persistence property **datanucleus.query.flushBeforeExecution** (defaults to *false*) and it will apply to all queries. Alternatively, to do this on a per query basis you would do

```
query.extension("datanucleus.query.flushBeforeExecution","true");
```

Controlling the execution : FetchPlan

When a Query is executed it executes in the datastore, which returns a set of results. DataNucleus could clearly read all results from this ResultSet in one go and return them all to the user, or could

allow control over this fetching process. JDO provides a *fetch size* on the Fetch Plan to allow this control. You would set this as follows

```
Query q = pm.newQuery(...);
q.getFetchPlan().setFetchSize(FetchPlan.FETCH_SIZE_OPTIMAL);
```

fetch size has 3 possible values.

- **FETCH_SIZE_OPTIMAL** - allows DataNucleus full control over the fetching. In this case DataNucleus will fetch each object when they are requested, and then when the owning transaction is committed will retrieve all remaining rows (so that the Query is still usable after the close of the transaction).
- **FETCH_SIZE_GREEDY** - DataNucleus will read all objects in at query execution. This can be efficient for queries with few results, and very inefficient for queries returning large result sets.
- **A positive value** - DataNucleus will read this number of objects at query execution. Thereafter it will read the objects when requested.

In addition to the number of objects fetched, you can also control which fields are fetched for each object of the candidate type. This is controlled via the *FetchPlan*.

For RDBMS any single-valued member will be fetched in the original SQL query, but with multiple-valued members this is not supported. However what will happen is that any collection/array field will be retrieved in a single SQL query for all candidate objects (by default using an EXISTS subquery); this avoids the "N+1" problem, resulting in 1 original SQL query plus 1 SQL query per collection member. Note that you can disable this by either not putting multi-valued fields in the FetchPlan, or by setting the query extension **datanucleus.rdbms.query.multivaluedFetch** to *none* (default is "exists" using the single SQL per field).

For non-RDBMS datastores the collection/map is stored by way of a Collection of ids of the related objects in a single "column" of the object and so is retrievable in the same query. See also Fetch Groups.

Extension: Load results at commit

DataNucleus also allows an extension to give further control. As mentioned above, when the transaction containing the Query is committed, all remaining results are read so that they can then be accessed later (meaning that the query is still usable). Where you have a large result set and you don’t want this behaviour you can turn it off by specifying a Query extension

```
q.extension("datanucleus.query.loadResultsAtCommit", "false");
```

so when the transaction is committed, no more results will be available from the query.

Extension: Ignore FetchPlan

In some situations you don't want all *FetchPlan* fields retrieving, and DataNucleus provides an extension to turn this off, like this

```
q.extension("datanucleus.query.useFetchPlan", "false");
```

ignoreCache(), setIgnoreCache()

The ignoreCache option setting specifies whether the query should execute entirely in the back end, instead of in the cache. If this flag is set to *true*, DataNucleus may be able to optimize the query execution by ignoring changed values in the cache. For optimistic transactions, this can dramatically improve query response times.

```
q.ignoreCache(true);
```

Control over locking of fetched objects

JDO allows control over whether objects found by a query are locked during that transaction so that other transactions can't update them in the meantime. To do this you would do

```
Query q = pm.newQuery(...);
q.serializeRead(true);
```

You can also specify this for all queries for all PMs using the persistence property **datanucleus.SerializeRead**. In addition you can perform this on a per-transaction basis by doing

```
tx.setSerializeRead(true);
```

If the datastore in use doesn't support locking of objects then this will do nothing

Timeout on query execution for reads

```
q.datastoreReadTimeoutMillis(1000);
```

Sets the timeout for this query (in milliseconds). Will throw a JDOUnsupportedOperationException if the query implementation doesn't support timeouts (for the current datastore).

Timeout on query execution for writes

```
q.datastoreWriteTimeoutMillis(1000);
```

Sets the timeout for this query (in milliseconds) when it is a delete/update. Will throw a JDOUnsupportedOperationException if the query implementation doesn't support timeouts (for the current datastore).

Extension: Loading Large Result Sets 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 extension **datanucleus.query.loadResultsAtCommit** to *false*.

To do this on a per query basis you would do

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

Extension: 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 extension **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
- *hard* : 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.addExtension("datanucleus.query.resultCacheType", "weak");
```

Extension: Size of Large Result Sets

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

Extension: Type of Result Set (RDBMS)

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.addExtension("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.

Extension: Result Set Control (RDBMS)

DataNucleus RDBMS provides a useful extension allowing control over the ResultSet's that are created by queries. Some properties are available 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.addExtension("datanucleus.rdbms.query.fetchDirection", "forward");
query.addExtension("datanucleus.rdbms.query.resultSetConcurrency", "read-only");
```

Alternatively you can specify these as persistence properties so that they apply to all queries for that PMF. 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.

JDOQL

JDO provides its own object-based query language (JDOQL), designed to have the power of SQL queries, yet retaining the Java object relationship that exist in the developers application model.

A JDOQL query may be created in several ways. Here's an example expressed in the 3 supported ways

```
// String-based JDOQL :
Query q = pm.newQuery("SELECT FROM mydomain.Person WHERE lastName == 'Jones' && age <
age_limit PARAMETERS int age_limit");
List<Person> results = (List<Person>)q.execute(20);

// Declarative JDOQL :
Query q = pm.newQuery(Person.class);
q.setFilter("lastName == 'Jones' && age < age_limit");
q.declareParameters("int age_limit");
List<Person> results = q.setParameters(20).executeList();

// Typed JDOQL :
JDOQLTypedQuery<Person> tq = pm.newJDOQLTypedQuery(Person.class);
QPerson cand = QPerson.candidate();
List<Person> results =
    tq.filter(cand.lastName.eq("Jones").and(cand.age.lt(tq.intParameter(
"age_limit"))))
        .setParameter("age_limit", "20").executeList();
```

So here in our example we select all "Person" objects with surname of "Jones" and where the persons age is below 20. The language is intuitive for Java developers, and is intended as their interface to accessing the persisted data model. As can be seen above, the query is made up of distinct parts: the class being processed (equates to the FROM clause in SQL), the data being selected (the SELECT clause in SQL), the filter (the WHERE clause in SQL), together with any sorting (the ORDER BY clause in SQL), etc.

We will cover the *string-based* and *declarative* modes of JDOQL API in this chapter, and the *Typed JDOQL* is covered in its own chapter.

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**.

JDOQL Single-String syntax

JDOQL queries can be defined in a single-string form, as follows

```
SELECT [UNIQUE] [<result>] [INTO <result-class>]
        [FROM <candidate-class> [EXCLUDE SUBCLASSES]]
        [WHERE <filter>]
        [VARIABLES <variable declarations>]
        [PARAMETERS <parameter declarations>]
        [<import declarations>]
        [GROUP BY <grouping>]
        [ORDER BY <ordering>]
        [RANGE <start>, <end>]
```

The "keywords" in the query are shown in UPPER CASE but can be in *UPPER* or *lower* case (but not MiXeD case). So giving an example

```
SELECT UNIQUE FROM mydomain.Employee ORDER BY departmentNumber
```

Candidate Class

By default the candidate "class" with JDOQL has to be a persistable class. This can then be referred to in the query using the *this* keyword (just like in Java). Also by default your query will return instances of subclasses of the candidate class. You can restrict to just instances of the candidate by specifying to exclude subclasses (see EXCLUDE SUBCLASSES in the string-based syntax, or by *setSubclasses(false)* when using the declarative API).

The "candidate" has an implicit "alias" in JDOQL, which is *this* (just like in Java). So in the rest of the query you can refer to a field of the candidate as *this.{fieldName}*

If the candidate has a table using a discriminator, the generated SQL for RDBMS will include a restriction of the possible discriminator values to the candidate and any applicable subclasses. If you want to override this and NOT have a discriminator restriction imposed in the SQL then you provide the query extension **datanucleus.query.dontRestrictDiscriminator** set to *true*.

Candidate Persistent Interface

DataNucleus also allows you to specify a candidate class as persistent interface. This is used where we want to query for instances of implementations of the interface. Let's take an example. We have an interface, and some implementations

```
@PersistenceCapable
public interface ComputerPeripheral
{
    @PrimaryKey
    long getId();
    void setId(long val);

    @Persistent
    String getManufacturer();
    void setManufacturer(String name);

    @Persistent
    String getModel();
    void setModel(String name);
}

@PersistenceCapable
public class Mouse implements ComputerPeripheral {...}

@PersistenceCapable
public class Keyboard implements ComputerPeripheral {...}
```

So we have made our interface persistable, and defined the identity property(ies) there. The implementations of the interface will use the identity defined in the interface. To query it we simply do

```
Query q = pm.newQuery(ComputerPeripheral.class);
List<ComputerPeripheral> results = q.executeList();
```

The key rules are

- You must define the interface as persistent
- The interface must define the identity/primary key member(s)
- The implementations must have the same definition of identity and primary key

Filter

The most important thing to remember when defining the *filter* for JDOQL is that **think how you would write it in Java, and its likely the same**. The *filter* has to be a boolean expression, and can include the candidate, fields/properties, literals, methods, parameters, variables, operators, instanceof, subqueries and casts.

With the Declarative API you would define the filter using the *Query.filter* method, like this

```
q.filter("this.inventory.name == 'MyInventory'");
```

Fields/Properties

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

```
price < 150.0
```

Note that, just like in Java, if you want to refer to a field/property of the candidate you can prefix the field by its implicit alias *this*

```
this.price < 150.0
```

You can also chain field references, so if you have a candidate class Product with a field of (persistable) type Inventory, which has a field *name*, then you could do

```
this.inventory.name == 'Backup'
```

In addition to the persistent fields, you can also access "public static final" fields of any class. You can do this as follows

```
taxPercent < mydomain.Product.TAX_BAND_A
```

So this will find all products that include a tax percentage less than some "BAND A" level. Where you are using "public static final" fields you can either fully-qualify the class name or you can include it in the "imports" section of the query (see later).

With JDOQL you do not do *explicit* joins. You instead use the fields/properties and navigate to the object you want to make use of in your query

With 1-1/N-1 relations this is simply a reference to the field/property, and place some restriction on it, like this

```
this.inventory.name == 'MyInventory'
```

With 1-N/M-N relations you would introduce a JDOQL variable and use something like

```
containerField.contains(elemVar)
```

and thereafter refer to *elemVar* for the element in the collection to place restrictions on the element. Similarly you can use *elemVar* in the result clause

RDBMS : By default when you navigate through a 1-1/N-1 relation in JDOQL DataNucleus will decide to join using either LEFT OUTER JOIN or INNER JOIN based on whether the relation is *nullable*. If it is nullable then LEFT OUTER JOIN will be used. You can change this default by specifying the persistence property (to apply to all queries) or query extension **datanucleus.query.jdoql.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.jdoql.navigationJoinTypeForFilter** and set it to either "INNERJOIN" or "LEFTOUTERJOIN".

Methods

When writing the "filter" for a JDOQL Query you can make use of some methods on the various Java types. The range of methods included as standard in JDOQL is not as flexible as with the true Java types, but the ones that are available are typically of much use. While DataNucleus supports all of the methods in the JDO standard, it also supports several yet to be standardised (extension) method. 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.

You can add "in-memory" evaluation support for other methods using this 

You can add "RDBMS datastore" support for other methods using this 

String Methods

Method	Description	Standard	In-Memory
startsWith(String)	Returns if the string starts with the passed string	✔	✔
startsWith(String, int)	Returns if the string starts with the passed string, from the passed position	✔	✔
endsWith(String)	Returns if the string ends with the passed string	✔	✔
indexOf(String)	Returns the first position of the passed string	✔	✔
indexOf(String,int)	Returns the position of the passed string, after the passed position	✔	✔
substring(int)	Returns the substring starting from the passed position	✔	✔

Method	Description	Standard	In-Memory
substring(int,int)	Returns the substring between the passed positions	✔	✔
toLowerCase()	Returns the string in lowercase	✔	✔
toUpperCase()	Retuns the string in UPPERCASE	✔	✔
matches(String pattern)	Returns whether string matches the passed expression. The pattern argument follows the rules of java.lang.String.matches method. Only the following regular expression patterns are required to be supported and are portable: global “(?i)” for case-insensitive matches; and “.” and “.*” for wild card matches. The pattern passed to matches must be a literal or parameter.	✔	✔
charAt(int)	Returns the character at the passed position	✔	✔
length()	Returns the length of the string	✔	✔
trim()	Returns a trimmed version of the string	✔	✔
concat(String)	Concatenates the current string and the passed string	✘	✔
equals(String)	Returns if the strings are equal	✘	✔
equalsIgnoreCase(String)	Returns if the strings are equal ignoring case	✘	✔
replaceAll(String, String)	Returns the string with all instances of *str1* replaced by *str2*	✘	✘
trimLeft()	Returns a trimmed version of the string (trimmed for leading spaces). **Only on RDBMS, Neo4j**	✘	✔
trimRight()	Returns a trimmed version of the string (trimmed for trailing spaces). **Only on RDBMS, Neo4j**	✘	✔

Here’s an example using a Product class, looking for objects which their abbreviation is the beginning of a trade name. The trade name is provided as parameter.

```
Declarative JDOQL :
Query query = pm.newQuery(Product.class);
query.setFilter(":tradeName.startsWith(this.abbreviation)");
List<Product> results = query.setParameters("Workbook Advanced").executeList();

Single-String JDOQL :
Query query = pm.newQuery("SELECT FROM mydomain.Product WHERE
:tradeName.startsWith(this.abbreviation)");
List results = (List)query.execute("Workbook Advanced");
```

Collection Methods

Method	Description	Standard	In-Memory
isEmpty()	Returns whether the collection is empty	✔	✔
contains(value)	Returns whether the collection contains the passed element	✔	✔
size()	Returns the number of elements in the collection	✔	✔
get(int)	Returns the element at that position of the List	✔	✔
indexOf(elem)	Returns the position in the List of the element.	✖	✔

Here's an example demonstrating use of contains(). We have an Inventory class that has a Collection of Product objects, and we want to find the Inventory objects with 2 particular Products in it. Here we make use of a variable (*prd* to represent the Product being contained

```
Declarative JDOQL :
Query query = pm.newQuery(Inventory.class);
query.setFilter("products.contains(prd) && (prd.name=='product 1' ||
prd.name=='product 2')");
List<Inventory> results = query.executeList();

Single-String JDOQL:
Query query = pm.newQuery("SELECT FROM mydomain.Inventory " +
    "WHERE products.contains(prd) && (prd.name=='product 1' || prd.name=='product
2')");
List results = (List)query.execute();
```

Map Methods

Method	Description	Standard	In-Memory
isEmpty()	Returns whether the map is empty	✔	✔
containsKey(key)	Returns whether the map contains the passed key	✔	✔
containsValue(value)	Returns whether the map contains the passed value	✔	✔
get(key)	Returns the value from the map with the passed key	✔	✔
size()	Returns the number of entries in the map	✔	✔
containsEntry(key, value)	Returns whether the map contains the passed entry	✖	✖

Here's an example using a Product class as a value in a Map. Our example represents an organisation that has several Inventories of products. Each Inventory of products is stored using a Map, keyed by the Product name. The query searches for all Inventories that contain a product with

the name "product 1".

```
Declarative JDOQL :
Query query = pm.newQuery(mydomain.Inventory.class, "products.containsKey('product
1')");
List<Inventory> results = query.execute();

Single-String JDOQL :
Query query = pm.newQuery("SELECT FROM mydomain.Inventory WHERE
products.containsKey('product 1')");
List results = (List)query.execute();
```

Here's the source code for reference

```
class Inventory
{
    Map<String, Product> products;
    ...
}
class Product
{
    ...
}
```

java.util.Date Temporal Methods

Method	Description	Standard	In-Memory
getDate()	Returns the day (of the month) for the date (java.util.Date types) in the timezone it was stored	✔	✔
getMonth()	Returns the month for the date (java.util.Date types) (0-11) in the timezone it was stored	✔	✔
getYear()	Returns the year for the date (java.util.Date types) in the timezone it was stored	✔	✔
getHour()	Returns the hour for the time (java.util.Date types) in the timezone it was stored	✔	✔
getMinute()	Returns the minute for the time (java.util.Date types) in the timezone it was stored	✔	✔
getSecond()	Returns the second for the time (java.util.Date types) in the timezone it was stored	✔	✔
getDayOfWeek()	Returns the day of the week for the date (java.util.Date types) (1-7) in the timezone it was stored	✔	✔

java.time Temporal Methods

Class	Method	Description	Standard	In-Memory
LocalDate	getDayOfMonth()	Returns the day (of the month) for the date (1-31) in the timezone it was stored	✔	✔
LocalDate	getDayOfWeek()	Returns the day of the week for the date (1-7) in the timezone it was stored	✔	✔
LocalDate	getMonthValue()	Returns the month for the date (1-12) in the timezone it was stored	✔	✔
LocalDate	getYear()	Returns the year for the date in the timezone it was stored	✔	✔
LocalDateTime	getDayOfMonth()	Returns the day (of the month) for the date in the timezone it was stored	✔	✔
LocalDateTime	getDayOfWeek()	Returns the day of the week for the date (1-7) in the timezone it was stored	✔	✔
LocalDateTime	getMonthValue()	Returns the month for the date (1-12) in the timezone it was stored	✔	✔
LocalDateTime	getYear()	Returns the year for the date in the timezone it was stored	✔	✔
LocalDateTime	getHour()	Returns the hour for the time in the timezone it was stored	✔	✔
LocalDateTime	getMinute()	Returns the minute for the time in the timezone it was stored	✔	✔
LocalDateTime	getSecond()	Returns the second for the time in the timezone it was stored	✔	✔
LocalTime	getHour()	Returns the hour for the time in the timezone it was stored	✔	✔
LocalTime	getMinute()	Returns the minute for the time in the timezone it was stored	✔	✔
LocalTime	getSecond()	Returns the second for the time in the timezone it was stored	✔	✔
MonthDay	getMonthValue()	Returns the month (1-12)	✔	✔
MonthDay	getDayOfMonth()	Returns the day of the month (1-31)	✔	✔
Period	getDays()	Returns the number of days	✔	✔
Period	getMonths()	Returns the number of months	✔	✔
Period	getYears()	Returns the number of years	✔	✔
YearMonth	getMonthValue()	Returns the month	✔	✔

Class	Method	Description	Standard	In-Memory
YearMonth	getYear()	Returns the year	✔	✔

Jodatime Temporal Methods

Class	Method	Description	Standard	In-Memory
Interval	getStart()	Returns the start of an Interval	✘	✔
Interval	getEnd()	Returns the end of an Interval	✘	✔

Enum Methods

Method	Description	Standard	In-Memory
ordinal()	Returns the ordinal of the enum (not implemented for enum expression when persisted as a string)	✔	✔
toString()	Returns the string form of the enum (not implemented for enum expression when persisted as a numeric)	✔	✔

Other Methods

Class	Method	Description	Standard	In-Memory
{}	length	Returns the length of an array. **Only on RDBMS**	✘	✘
{}	contains(object)	Returns true if the array contains the object. **Only on RDBMS**	✘	✘
java.util.Optional	isPresent()	Returns whether the value is present in this optional.	✔	✔
java.util.Optional	get()	Returns the delegated object	✔	✔
java.util.Optional	orElse(object)	Returns the value of the optional if present, otherwise the supplied object.	✔	✔

Static Methods

Method	Description	Standard	In-Memory
Math.abs(number)	Returns the absolute value of the passed number	✔	✔
Math.sqrt(number)	Returns the square root of the passed number	✔	✔
Math.cos(number)	Returns the cosine of the passed number	✔	✔
Math.sin(number)	Returns the absolute value of the passed number	✔	✔
Math.tan(number)	Returns the tangent of the passed number	✔	✔
Math.acos(number)	Returns the arc cosine of the passed number	✔	✔
Math.asin(number)	Returns the arc sine of the passed number	✔	✔
Math.atan(number)	Returns the arc tangent of the passed number	✔	✔
Math.ceil(number)	Returns the ceiling of the passed number	✔	✔
Math.exp(number)	Returns the exponent of the passed number	✔	✔
Math.floor(number)	Returns the floor of the passed number	✔	✔
Math.log(number)	Returns the log(base e) of the passed number	✔	✔
Math.round(number)	Returns the rounded value of the passed number	✘	✘
Math.toDegrees(number)	Returns the degrees of the passed radians value	✘	✔
Math.toRadians(number)	Returns the radians of the passed degrees value	✘	✔
Math.power(number, power)	Returns the passed number to the specified power	✘	✘
JDOHelper.getObjectId(object)	Returns the object identity of the passed persistent object	✔	✔
JDOHelper.getVersion(object)	Returns the version of the passed persistent object	✔	✔
SQL_rollup({object})	Perform a rollup operation over the results. **Only for some RDBMS e.g DB2, MSSQL, Oracle**	✘	✘
SQL_cube({object})	Perform a cube operation over the results. **Only for some RDBMS e.g DB2, MSSQL, Oracle**	✘	✘
SQL_boolean({sql})	Embed the provided SQL and return a boolean result. **Only on RDBMS**	✘	✘
SQL_numeric({sql})	Embed the provided SQL and return a numeric result. **Only on RDBMS**	✘	✘

Geospatial Methods

In terms of geospatial types that are part of the JRE

Class	Method	Description	Standard	In-Memory
java.awt.Point	getX()	Returns the X coordinate. **Only on RDBMS**	✗	✓
java.awt.Point	getY()	Returns the Y coordinate. **Only on RDBMS**	✗	✓
java.awt.Rectangle	getX()	Returns the X coordinate. **Only on RDBMS**	✗	✓
java.awt.Rectangle	getY()	Returns the Y coordinate. **Only on RDBMS**	✗	✓
java.awt.Rectangle	getWidth()	Returns the width. **Only on RDBMS**	✗	✓
java.awt.Rectangle	getHeight()	Returns the height. **Only on RDBMS**	✗	✓

In terms of geospatial types that are provided by more specialised libraries, such as JTS, the following applies.

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 methods 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 methods. The following tables list all available methods as well as information about which RDBMS implement them. An entry in the "Result" column indicates, whether the function may be used in the result part of a JDOQL query.

Methods on Type Geometry (OGC SF 3.2.10)

Method	Description	Result	PostGIS	MySQL	Oracle Spatial
getDimension()	Returns the dimension of the Geometry.	✓	✓	✓	✓
getGeometryType()	Returns the name of the instantiable subtype of Geometry.	✓	✓	✓	✓
getSRID()	Returns the Spatial Reference System ID for this Geometry.	✓	✓	✓	✓
isEmpty()	TRUE if this Geometry corresponds to the empty set.	! [1]	✓	✓	✓

Method	Description	Result	PostGIS	MySQL	Oracle Spatial
isSimple()	TRUE if this Geometry is simple, as defined in the Geometry Model.	❗ [1]	✔	✔	✔
getBoundary()	Returns a Geometry that is the combinatorial boundary of the Geometry.	✔	✔	✔	✔
getEnvelope()	Returns the rectangle bounding Geometry as a Polygon.	✔	✔	✔	✔
toText()	Returns the well-known textual representation.	✔	✔	✔	✔
toBinary()	Returns the well-known binary representation.	✖	✔	✔	✔

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

Methods on Type Point (OGC SF 3.2.11)

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

Methods on Type Curve (OGC SF 3.2.12)

Method	Description	Result	PostGIS	MySQL	Oracle Spatial
getStartPoint()	Returns the first point of the Curve.	✔	✔	✔	✔
getEndPoint()	Returns the last point of the Curve.	✔	✔	✔	✔
isRing()	Returns TRUE if Curve is closed and simple.	❗ [1]	✔	✔	✔

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

Methods on Type Curve / MultiCurve (OGC SF 3.2.12, 3.2.17)

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

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

Methods on Type LineString (OGC SF 3.2.13)

Method	Description	Result [1]	PostGIS	MySQL	Oracle Spatial
getNumPoints()	Returns the number of points in the LineString.	✔	✔	✔	✔
getPointN(Integer)	Returns Point n.	✔	✔	✔	✔

Methods on Type Surface / MultiSurface (OGC SF 3.2.14, 3.2.18)

Method	Description	Result	PostGIS	MySQL	Oracle Spatial
getCentroid()	Returns the centroid of Surface/MultiSurface, which may lie outside of it.	✔	✔	✘ [1]	✔
getArea()	Returns the area of Surface/MultiSurface.	✔	✔	✔	✔
getPointOnSurface()	Returns a Point guaranteed to lie on the surface.	✔	✔	✘ [1]	✔ [2]

[1] MySQL does not implement these methods. [2] Oracle takes an argument to this method (see Oracle docs)

Methods on Type Polygon (OGC SF 3.2.15)

Method	Description	Result	PostGIS	MySQL	Oracle Spatial
getExteriorRing()	Returns the exterior ring of Polygon.	✔	✔	✔	✔
getNumInteriorRing()	Returns the number of interior rings.	✔	✔	✔	✔
getInteriorRingN(Integer)	Returns the nth interior ring.	✔	✔	✔	✔

Methods on Type GeomCollection (OGC SF 3.2.16)

Method	Description	Result	PostGIS	MySQL	Oracle Spatial
getNumGeometries()	Returns the number of geometries in the collection.	✔	✔	✔	✔
getGeometryN(Integer)	Returns the nth geometry in the collection.	✔	✔	✔	✔

Methods that test Spatial Relationships (OGC SF 3.2.19)

Method	Description	Result [1]	PostGIS	MySQL	Oracle Spatial
equals(Geometry)	TRUE if the two geometries are spatially equal.	❗	✔	❗ [2]	✔
disjoint(Geometry)	TRUE if the two geometries are spatially disjoint.	❗	✔	❗ [2]	✔
touches(Geometry)	TRUE if the first Geometry spatially touches the other Geometry.	❗	✔	❗ [2]	✔
within(Geometry)	TRUE if first Geometry is completely contained in second Geometry.	❗	✔	❗ [2]	✔
overlaps(Geometry)	TRUE if first Geometries is spatially overlapping the other Geometry.	❗	✔	❗ [2]	✔

Method	Description	Result [1]	PostGIS	MySQL	Oracle Spatial
crosses(Geometry)	TRUE if first Geometry crosses the other Geometry.	❗	✔	✔	✔
intersects(Geometry)	TRUE if first Geometry spatially intersects the other Geometry.	❗	✔	❗ [2]	✔
contains(Geometry)	TRUE if second Geometry is completely contained in first Geometry.	❗	✔	❗ [2]	✔
relate(Geometry, String)	TRUE if the spatial relationship specified by the patternMatrix holds.	❗	✔	✔	✔
bboxTest(Geometry)	Returns TRUE if if the bounding box of this Geometry overlaps the passed Geometry's bounding box	❗ [1]	✔	✔	✔

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

[2] MySQL does not implement these methods according to the specification. They return the same result as the corresponding MBR-based methods.

Methods on Distance Relationships (OGC SF 3.2.20)

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

[1] MariaDB 5.3.3+ implements this.

Methods that implement Spatial Operators (OGC SF 3.2.21)

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

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

Static Methods for Constructing a Geometry Value given its Well-known Representation (OGC SF 3.2.6, 3.2.7)

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

Method	Description	Result [1]	PostGIS	MySQL	Oracle Spatial
Spatial.mPolyFromText(String, Integer)	Construct a MultiPolygon given its well-known textual representation.	✗	✓	✓	✓
Spatial.geomCollFromText(String, Integer)	Construct a GeometryCollection given its well-known textual representation.	✗	✓	✓	✓
Spatial.geomFromWKB(String, Integer)	Construct a Geometry value given its well-known binary representation.	✗	✓	✓	✓
Spatial.pointFromWKB(String, Integer)	Construct a Point given its well-known binary representation.	✗	✓	✓	✓
Spatial.lineFromWKB(String, Integer)	Construct a LineString given its well-known binary representation.	✗	✓	✓	✓
Spatial.polyFromWKB(String, Integer)	Construct a Polygon given its well-known binary representation.	✗	✓	✓	✓
Spatial.mPointFromWKB(String, Integer)	Construct a MultiPoint given its well-known binary representation.	✗	✓	✓	✓
Spatial.mLineFromWKB(String, Integer)	Construct a MultiLineString given its well-known binary representation.	✗	✓	✓	✓
Spatial.mPolyFromWKB(String, Integer)	Construct a MultiPolygon given its well-known binary representation.	✗	✓	✓	✓
Spatial.geomCollFromWKB(String, Integer)	Construct a GeometryCollection given its well-known binary representation.	✗	✓	✓	✓

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

Supplementary Static Methods

These functions are only supported on certain RDBMS.

Method	Description	Result	PostGIS	MySQL	Oracle Spatial
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	✔	✔	✘	✘

Method	Description	Result	PostGIS	MySQL	Oracle Spatial
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.mbrEqual(Geometry, Geometry)	Returns 1 or 0 to indicate whether the minimum bounding rectangles of the two geometries g1 and g2 are the same.	✔	✘	✔	✘
MySQL.mbrDisjoint(Geometry, Geometry)	Returns 1 or 0 to indicate whether the minimum bounding rectangles of the two geometries g1 and g2 are disjoint (do not intersect).	✔	✘	✔	✘
MySQL.mbrIntersects(Geometry, Geometry)	Returns 1 or 0 to indicate whether the minimum bounding rectangles of the two geometries g1 and g2 intersect.	✔	✘	✔	✘
MySQL.mbrTouches(Geometry, Geometry)	Two geometries spatially touch if their interiors do not intersect, but the boundary of one of the geometries intersects either the boundary or the interior of the other.	✔	✘	✔	✘
MySQL.mbrWithin(Geometry, Geometry)	Returns 1 or 0 to indicate whether the minimum bounding rectangle of g1 is within the minimum bounding rectangle of g2.	✔	✘	✔	✘
MySQL.mbrContains(Geometry, Geometry)	Returns 1 or 0 to indicate whether the minimum bounding rectangle of g1 contains the minimum bounding rectangle of g2.	✔	✘	✔	✘
MySQL.mbrOverlaps(Geometry, Geometry)	Two geometries spatially overlap if they intersect and their intersection results in a geometry of the same dimension but not equal to either of the given geometries.	✔	✘	✔	✘
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.	✔	✘	✘	✔

Method	Description	Result	PostGIS	MySQL	Oracle Spatial
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.	✔	✖	✖	✔

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

Examples

The following sections provide some examples of what can be done using spatial methods in JDOQL 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;
    }
    ....
}
```

```
<jdo>
    <package name="mydomain.samples.pggeometry">
        <extension vendor-name="datanucleus" key="spatial-dimension" value="2"/>
        <extension vendor-name="datanucleus" key="spatial-srid" value="4326"/>

        <class name="SampleLineString" table="samplepglinestring" detachable="true">
            <field name="id"/>
            <field name="name"/>
            <field name="geom" persistence-modifier="persistent">
                <extension vendor-name="datanucleus" key="mapping" value="no-
userdata"/>
            </field>
        </class>
    </package>
</jdo>
```

Example 1 - Spatial Method in the Filter of a Query

This example shows how to use spatial methods 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.

```
Double limit = new Double(100.0);
Query query = pm.newQuery(SampleLineString.class, "geom != null && geom.length() <
:limit");
List list = (List) query.execute(limit);
```

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

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

```
query = pm.newQuery(SampleLineString.class, "id == :id");
query.setResult("geom.pointN(2)");
query.setUnique(true);
Geometry point = (Geometry) query.execute(new Long(1001));
```

Example 3 - Nested Methods

You may want to use nested methods 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 query = pm.newQuery(SampleLineString.class, "geom != null &&
Spatial.equals(geom.endPoint(), :point)");
List list = (List) query.execute(point);
```

Literals

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

```
Query q = pm.newQuery("SELECT FROM mydomain.Person WHERE age == 25");
```

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

```
Query q = pm.newQuery("SELECT FROM mydomain.Person WHERE firstName == 'John'");
```

DataNucleus also provides an extension to support array literals for RDBMS. You would do as follows in your JDOQL

```
Query q = pm.newQuery("SELECT FROM mydomain.Person WHERE {'John', 'Fred',
'Graham'}.contains(firstName)");
```

namely using curly brackets to represent the array with its literal elements.

RDBMS : Parameters .v. Literals

When considering whether to embody a literal into a JDOQL 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

With a query you can pass values into the query as parameters. This is useful where you don't want to embed particular values in the query itself, so making it reusable with different values. JDOQL allows two types of parameters.

Explicit Parameters

If you *declare* the parameters when defining the query (using the PARAMETERS keyword in the single-string form, or via the declareParameters method) then these are **explicit** parameters. This sets the type of the parameter, and when you pass the value in at *execute* it has to be of that type. For example

```
Query query = pm.newQuery("SELECT FROM mydomain.Product WHERE price < limit PARAMETERS
double limit");
List results = (List)query.execute(150.00);
```

Note that if declaring multiple parameters then they should be comma-separated.

With the Declarative API you would define explicit parameters like this (and use comma-separated if defining multiple)

```
q.parameters("double limit");
```

Implicit Parameters

If you don't declare the parameters when defining the query but instead prefix identifiers in the query with **:** (colon) then these are **implicit** parameters. For example

```
Query query = pm.newQuery("SELECT FROM mydomain.Product WHERE price < :limit");
List results = (List)query.execute(150.00);
```

In some situations you may have a map of parameters keyed by their name, yet the query in question doesn't need all parameters. Normal JDO execution would throw an exception here since they are inconsistent with the query. You can omit this check by setting

```
q.addExtension("datanucleus.query.checkUnusedParameters", "false");
```

Setting parameters at execution time

Defining a query to accept parameters is only the first part. You then need to specify the parameter values at execution time. This can be done in many ways, but here are some examples

```
// === JDOQL with named parameters ===
...
q.setFilter("this.name == :name && this.serialNo == :serial");

Map params = new HashMap();
params.put("name", "Walkman");
params.put("serial", "123021");


// Set parameter values via method call
q.setNamedParameters(params);


// Alternatively set the parameter values on execute()
q.executeWithMap(params);



// === JDOQL with numbered parameters ===
...
q.setFilter("this.name == ?1 && this.serialNo == ?2");

// Set parameter values via method call, using the number order of the query
parameters
q.setParameters("Walkman", "123021");


// Alternatively set the parameter values on execute(), using the number order of the
query parameters
q.execute("Walkman", "123021");
```

Variables

In JDOQL you can connect two parts of a query using something known as a variable. For example, we want to retrieve all objects with a collection that contains a particular element, and where the element has a particular field value. We define a query like this

```
Query query = pm.newQuery("SELECT FROM mydomain.Supplier " +
    "WHERE this.products.contains(prod) && prod.name == 'Beans' VARIABLES
mydomain.Product prod");
```

So we have a variable in our query called *prod* that connects the two parts. You can declare your variables (using the VARIABLES keyword in the single-string form, or via the declareVariables method) if you want to define the type like here (**explicit**), or you can leave them for the query compilation to determine (**implicit**).

Another example, in this case our candidate (Product) has no relation, but a class (Inventory) has a

relation (1-N) to it (field "products") and we want to query based on that relation, returning the product name for a particular inventory.

```
Query q = pm.newQuery("SELECT this.name FROM mydomain.Product WHERE
inv.products.contains(this) AND inv.name == 'Sale' VARIABLES mydomain.Inventory inv");
```

Note that if declaring multiple variables then they should be semicolon-separated. See also this blog post which demonstrates variables across 1-1 "relations" where you only have the "id" stored rather than a real relation.

With the Declarative API you would define explicit variables like this

```
q.variables("mydomain.Product prod");
```

RDBMS : In all situations we aim for DataNucleus JDOQL implementation to work out the right way of linking a variable into the query, whether this is via a join (INNER, LEFT OUTER), or via a subquery. As you can imagine this can be complicated to work out the optimum for all situations so with that in mind we allow (for a limited number of situations) the option of specifying the join type. This is achieved by setting the query extension *datanucleus.query.jdoql.{varName}.join to the required type. For 1-1 relations this would be either "INNERJOIN" or "LEFTOUTERJOIN", and for 1-N relations this would be either "INNERJOIN", "LEFTOUTERJOIN" or "SUBQUERY".

Please, if you find a situation where the optimum join type is not chosen then report it in GitHub so it can be registered for future work

Imports

JDOQL uses the imports declaration to create a type namespace for the query. During query compilation, the classes used in the query, if not fully qualified, are searched in this namespace. The type namespace is built with primitives types, java.lang.* package, package of the candidate class, import declarations (if any).

To resolve a class, the JDOQL compiler will use the class fully-qualified name to load it, but if the class is not fully qualified, it will search by prefixing the class name with the imported package names declared in the type namespace. All classes loaded by the query must be acessible by either the candidate class classloader, the PersistenceManager classloader or the current Thread classloader. The search algorithm for a class in the JDOQL compiler is the following:

- if the class is fully qualified, load the class.
- if the class is not fully qualified, iterate each package in the type namespace and try to load the class from that package. This is done until the class is loaded, or the type namespace package names are exhausted. If the class cannot be loaded an exception is thrown.

Note that the search algorithm can be problematic in performance terms if the class is not fully qualified or declared in imports using package notation. To avoid such problems, either use fully qualified class names or import the class in the imports declaration.

If you always fully-qualify the candidate, variable and parameter types then there is no need to specify any *imports* (just like in Java).

IF ELSE expressions

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

```
SELECT p.personNum, IF (p.age < 18) 'Youth' ELSE IF (p.age >= 18 && p.age < 65)
'Adult' ELSE 'Old' FROM mydomain.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 JDOQL IF ELSE expression is

```
IF (conditional_expression) scalar_expression {ELSE IF (conditional_expression)
scalar_expression}* ELSE scalar_expression
```

Operators

The following list describes the operator precedence in JDOQL.

- Cast
- Unary ("~") ("!")
- Unary ("+") ("-")
- Multiplicative ("*") ("/") ("%")
- Additive ("+") ("-")
- Relational (">=") (">") ("⇐") ("<") ("instanceof")
- Equality ("==") ("!=")
- Boolean logical AND ("&")
- Boolean logical OR ("|")
- DataNucleus Extension : Bitwise AND ("&") - for integral types on PostgreSQL, MySQL, SQLServer, NuoDB
- DataNucleus Extension : Bitwise OR ("|") - for integral types on PostgreSQL, MySQL, SQLServer, NuoDB
- DataNucleus Extension : Bitwise XOR ("^") - for integral types on PostgreSQL, MySQL, SQLServer, NuoDB
- Conditional AND ("&&")

- Conditional OR ("||")

The concatenation operator(+) concatenates a String to either another String or Number. Concatenations of String or Numbers to null results in null.

instanceof

JDOQL allows the Java keyword **instanceof** so you can compare objects against a class.

Let's take an example. We have a class A that has a field "b" of type B and B has subclasses B1, B2, B3. Clearly the field "b" of A can be of type B, B1, B2, B3 etc, and we want to find all objects of type A that have the field "b" that is of type B2. We do it like this

```
Declarative JDOQL :
Query query = pm.newQuery(A.class);
query.setFilter("b instanceof mydomain.B2");
List<A> results = query.executeList();

Single-String JDOQL :
Query query = pm.newQuery("SELECT FROM mydomain.A WHERE b instanceof mydomain.B2");
List results = (List)query.execute();
```

casting

JDOQL allows use of Java-style casting so you can type-convert fields etc.

Let's take an example. We have a class A that has a field "b" of type B and B has subclasses B1, B2, B3. The B2 subtype has a field "other", and we know that the filtered A will have a B2. You could specify a filter using the "B2.other" field like this

```
((mydomain.B2)b).other == :someVal"
```

Subqueries

With JDOQL 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. JDOQL allows subqueries, so that both calculations can be performed in one query. The syntax of a string-based subquery is as follows

```
SELECT <subquery-result-clause>
        [FROM <subquery-from-clause>
        [WHERE <filter>]
        [VARIABLES <variable declarations>]
        [PARAMETERS <parameter declarations>]
```

The *subquery-result-clause* consists of an optional keyword "DISTINCT" followed by a single expression. The *subquery-from-clause* may have one of two forms: A candidate class name followed by an optional alias definition followed by an optional "EXCLUDE SUBCLASSES", or a field access expression followed by an optional alias definition.

Here's an example, using single-string JDOQL

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

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

We can specify the same query using the Declarative API, like this

```
Query averageSalaryQuery = pm.newQuery(Employee.class);
averageSalaryQuery.setResult("avg(this.salary)");

Query q = pm.newQuery(Employee.class, "salary > averageSalary");
q.declareVariables("double averageSalary");
q.addSubquery(averageSalaryQuery, "double averageSalary", null, null);
List<Employee> results = q.executeList();
```

So we define a subquery as its own Query (that could be executed just like any query if so desired), and the in the main query have an implicit variable that we define as being represented by the subquery.

Referring to the outer query in the subquery

JDOQL subqueries allows use of the outer query fields within the subquery if so desired. Taking the above example and extending it, here is how we do it in single-string JDOQL

```
SELECT FROM mydomain.Employee WHERE salary >
    (SELECT avg(e.salary) FROM mydomain.Employee e WHERE e.lastName == this.lastName)
```

So with single-string JDOQL we make use of the alias identifier "this" to link back to the outer query.

Using the Declarative API, to achieve the same thing we would do

```
Query averageSalaryQuery = pm.newQuery(Employee.class);
averageSalaryQuery.setResult("avg(this.salary)");
averageSalaryQuery.setFilter("this.lastName == :lastNameParam");

Query q = pm.newQuery(Employee.class, "salary > averageSalary");
q.declareVariables("double averageSalary");
q.addSubquery(averageSalaryQuery, "double averageSalary", null, "this.lastName");
List<Employee> results = q.executeList();
```

So with the Declarative API we make use of parameters, and the last argument to *addSubquery* is the value of the parameter *lastNameParam*.

Candidate of the subquery being part of the outer query

There are occasions where we want the candidate of the subquery to be part of the outer query, so JDOQL subqueries has the notion of a *candidate expression*. This is an expression relative to the candidate of the outer query. An example

```
SELECT FROM mydomain.Employee WHERE this.weeklyhours >
    (SELECT AVG(e.weeklyhours) FROM this.department.employees e)
```

so the candidate of the subquery is *this.department.employees*. If using a candidate expression we must provide an alias. You can do the same with the Declarative API. Like this

```
Query averageHoursQuery = pm.newQuery(Employee.class);
averageHoursQuery.setResult("avg(this.weeklyhours)");

Query q = pm.newQuery(Employee.class);
q.setFilter("this.weeklyhours > averageWeeklyhours");
q.addSubquery(averageHoursQuery, "double averageWeeklyhours",
"this.department.employees", null);
```

so now our subquery has a candidate related to the outer query candidate.

In strict JDOQL you can only have the subquery in the "filter" (WHERE) clause. DataNucleus additionally allows it in the "result" (SELECT) clause.

Using methods on the subquery

A subquery is effectively a Collection, so you have access to the normal methods of a Collection to use on the subquery. Here are a couple of examples

```
SELECT FROM mydomain.Manager WHERE (SELECT FROM mydomain.Employee e WHERE e.manager ==
this).isEmpty()
```

which returns all *Manager* objects which have no *Employee(s)*.

This can equally be expressed using *contains()*

```
SELECT FROM mydomain.Manager WHERE !(SELECT FROM mydomain.Employee e).contains(this)
```

There is no *size()* method on subqueries but you can achieve the same by selecting `COUNT(e)` in the subquery (where `e` is the subquery alias).

Result clause

By default (when not specifying the result) the objects returned will be of the candidate class type, where they match the query filter. The *result* clause can contain (any of) the following

- DISTINCT - optional keyword at the start of the results to make them distinct
- *this* - the candidate instance
- A field name
- A variable
- A parameter (though why you would want a parameter returning is hard to see since you input the value in the first place)
- An aggregate (count(), avg(), sum(), min(), max())
- An expression involving a field (e.g "field1 + 1")
- A navigational expression (navigating from one field to another ... e.g "field1.field4")

so you could specify something like

```
count(field1), field2
```

There are situations when you want to return a single number for a column, representing an aggregate of the values of all records. There are 5 standard JDO aggregate functions available. These are

- **avg(val)** - returns the average of "val". "val" can be a field, numeric field expression or "distinct field". Returns double.
- **sum(val)** - returns the sum of "val". "val" can be a field, numeric field expression, or "distinct field". Returns the same type as the type being summed
- **count(val)** - returns the count of records of "val". "val" can be a field, or can be "this", or "distinct field". Returns long
- **min(val)** - returns the minimum of "val". "val" can be a field. Returns the same type as the type used in "min"
- **max(val)** - returns the maximum of "val". "val" can be a field. Returns the same type as the type used in "max"

So to utilise these you could specify a result like

```
max(price), min(price)
```

This will return a single row of results with 2 values, the maximum price and the minimum price.

Note that what you specify in the *result* defines what form of result you get back when executing the query.

- **{ResultClass}** - this is returned if you have only a single row in the results and you specified a result class.
- **Object** - this is returned if you have only a single row in the results and a single column. This is achieved when you specified either UNIQUE, or just an aggregate (e.g "max(field2)")
- **Object[]** - this is returned if you have only a single row in the results, but more than 1 column (e.g "max(field1), avg(field2)")
- **List<{ResultClass}>** - this is returned if you specified a result class.
- **List<Object>** - this is returned if you have only a single column in the result, and you don't have only aggregates in the result (e.g "field2")
- **List<Object[]>** - this is returned if you have more than 1 column in the result, and you don't have only aggregates in the result (e.g "field2, avg(field3)")

With the string-based API the *result* is part of the query. With the Declarative API you would specify the result like this

```
q.result(result);

// alternatively q.setResult(result)
```

Result Class

By default a JDOQL query will return a result matching the result clause. You can override this if you wish by specifying a result class. If your query has only a single row in the results then you will get an object of your result class back, otherwise you get a List of result class objects. The *Result Class* has to meet certain requirements. These are

- Can be one of Integer, Long, Short, Float, Double, Character, Byte, Boolean, String, java.math.BigInteger, java.math.BigDecimal, java.util.Date, java.sql.Date, java.sql.Time, java.sql.Timestamp, or Object[]
- Can be a user defined class, that has either a constructor taking arguments of the same type as those returned by the query (in the same order), or has a public put(Object, Object) method, or public setXXX() methods, or public fields.

In terms of how the *Result Class* looks, you have two options

- 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 Price
{
    protected double amount = 0.0;
    protected String currency = null;

    public Price(double amount, String currency)
    {
        this.amount = amount;
        this.currency = currency;
    }

    ...
}
```

- 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 Price
{
    protected double amount = 0.0;
    protected String currency = null;

    public Price()
    {
    }

    public void setAmount(double amt) {this.amount = amt;}
    public void setCurrency(String curr) {this.currency = curr;}

    ...
}
```

With the string-based API the *resultClass* can be part of the query (*INTO {result-class}*). With the Declarative API you would specify the result class upon execution

```
q.executeResultList(MyResultClass.class);

q.executeResultUnique(MyResultClass.class);
```

Grouping of Results

By default your results will have no specified "grouping". You can specify a *grouping* to include an optional *having* expression. When grouping is specified, each result expression must either be an expression contained in the grouping, or an aggregate evaluated once per group.

With the string-based API the *grouping* would be part of the query (*GROUP BY {grouping}*). With the Declarative API you would specify the grouping like this

```
q.groupBy(grouping);
```

Ordering of Results

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.

In the ordering you can also define 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]
```

Note that this is only supported for a few RDBMS (H2, HSQLDB, PostgreSQL, DB2, Oracle, Derby, Firebird, SQLServer v11+).

With the string-based API the *ordering* would be part of the query (*ORDER BY {ordering}*). With the Declarative API you would specify the ordering like this

```
q.orderBy(ordering);
```

Range of Results

By default your query will return all results matching the specified filter. You can restrict which results are returned by using the *range*. For example

```
RANGE 10,20
```

which will return just the results numbers 10-19 inclusive. Obviously bear in mind that if specifying the range then you should really specify an ordering otherwise the range positions will be not defined.

With the string-based API the *ordering* would be part of the query (*RANGE {start,end}*). With the Declarative API you would specify the range like this

```
q.range(10, 20);
```

RANGE handling is implemented efficiently for MySQL, Postgresql, HSQLDB, H2, SQLServer (using the LIMIT SQL keyword) and Oracle (using the ROWNUM keyword), with the query only finding the objects required by the user directly in the datastore. For other RDBMS the query will retrieve all objects up to the "to" record, and will not pass any unnecessary objects that are before the "from" record.

JDOQL In-Memory queries

The typical use of a JDOQL query is to translate it into the native query language of the datastore and return objects matched by the query. Sometimes you want to query over a set of objects that you have to hand, or for some datastores it is simply impossible to support the full JDOQL syntax in the datastore *native query language*. In these situation we need to evaluate the query *in-memory*. In the latter case of the datastore not supported the full JDOQL syntax 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.

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

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

This is also useful where you have a Collection of (persisted) objects and want to run a query over the Collection. Simply turn on in-memory evaluation, and supply the candidate collection to the query, and no communication with the datastore will be needed.

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

Specify candidates to query over

With JDO you can define a set of candidate objects that should be queried, rather than just going to the datastore to retrieve those objects. When you specify this you will automatically be switched to evaluate the query in-memory. You set the candidates like this

```
Query query = pm.newQuery(...);
query.setCandidates(myCandidates);
List<Product> results = query.executeList();
```

Update/Delete queries

JDOQL offers some possibilities for updating/deleting data in the datastore via query. Note that only the first of these is standard JDOQL, whereas the others are DataNucleus extensions.

Deletion by Query

If you want to delete instances of a candidate using a query, you simply define the query candidate/filter in the normal way, and then instead of calling *query.executeXXX* you call *query.deletePersistentAll()*. Like this

```
Query query = pm.newQuery("SELECT FROM mydomain.A WHERE this.value < 50");
Long number = (Long)query.deletePersistentAll();
```

The value returned is the number of instances that were deleted. Note that this will perform any cascade deletes that are defined for these instances. In addition, all instances in memory will reflect this deletion.

Bulk Delete

DataNucleus provides an extension to allow bulk deletion. This differs from the "Deletion by Query" above in that it simply goes straight to the datastore and performs a bulk delete, leaving it to the datastore referential integrity to handle relationships. To enable "bulk delete" you need the persistence property **datanucleus.query.jdoql.allowAll** set to *true*. You then perform "bulk delete" like this

```
Query query = pm.newQuery("DELETE FROM mydomain.A WHERE this.value < 50");
Long number = (Long)query.execute();
```

Again, the number returned is the number of records deleted.

Bulk Delete will not be reflected in L1 cached objects, and cascading defined in metadata will not be invoked when using this

Bulk Update

DataNucleus provides an extension to allow bulk update. This allows you to do bulk updates direct to the datastore without having to load objects into memory etc. To enable "bulk update" you need the persistence property **datanucleus.query.jdoql.allowAll** set to *true*. You then perform "bulk update" like this

```
Query query = pm.newQuery("UPDATE mydomain.A SET this.value=this.value-5.0 WHERE
this.value > 100");
Long number = (Long)query.execute();
```

Again, the number returned is the number of records updated.

Bulk Update will not be reflected in L1 cached objects, and cascading defined in metadata will not be invoked when using this

JDOQL Strictness

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

```
Query query = pm.newQuery(...);
query.addExtension("datanucleus.query.jdoql.strict", "true");
```

JDOQL : SQL Generation for RDBMS

When using the method *contains* on a collection (or *containsKey*, *containsValue* on a map) this will either add an EXISTS subquery (if there is a NOT or OR present in the query) or will add an INNER JOIN across to the element table. Let's take an example

```
SELECT FROM mydomain.A
WHERE (elements.contains(b1) && b1.name == 'Jones')
VARIABLES mydomain.B b1
```

Note that we add the *contains* first that binds the variable "b1" to the element table, and then add the condition on the variable. The order is important here. If we instead had put the condition on the variable first we would have had to do a CROSS JOIN to the variable table and then try to repair the situation and change it to INNER JOIN if possible. In this case the generated SQL will be like

```
SELECT `A0`.`ID`
FROM `A` `A0`
INNER JOIN `B` `B0` ON `A0`.ID = `B`.ELEMENT
WHERE `B0`.NAME = 'Jones'
```

JDOQL Typed

JDO 3.2 introduces a way of performing queries using a JDOQLTypedQuery API, that copes with refactoring of classes/fields. The API follows the same JDOQL syntax that we have seen earlier in terms of the components of the query etc. It produces queries that are much more elegant and simpler than the equivalent "Criteria" API in JPA, or the Hibernate Criteria API. See this comparison of JPA Criteria and JDO Typesafe which compares a prototype of this JDOQLTypedQuery API against JPA Criteria.

Preparation

To set up your environment to use this JDOQLTypedQuery API you need to enable annotation processing, place some DataNucleus jars in your build path, and specify a `@PersistenceCapable` annotation on your classes to be used in queries (you can still provide the remaining information in XML metadata if you wish to). This annotation processor will (just before compile of your persistable classes), create a query metamodel "Q" class for each persistable class. This is similar step to what QueryDSL requires, or indeed the JPA Criteria static metamodel.

Using Maven

With Maven you need to have the following in your POM

```
<dependencies>
    <dependency>
        <groupId>org.datanucleus</groupId>
        <artifactId>datanucleus-jdo-query</artifactId>
        <version>[5.0.9, )</version>
    </dependency>
    <dependency>
        <groupId>org.datanucleus</groupId>
        <artifactId>javax.jdo</artifactId>
        <version>[3.2.0-m9, 3.9)</version>
    </dependency>
    ...
 </dependencies>
```

This creates the "metamodel" Q classes under *target/generated-sources/annotations/*. You can change this location using the configuration property **generatedSourcesDirectory** of the *maven-compiler-plugin*.

Using Eclipse

With Eclipse you need to

- Go to *Java Compiler* and make sure the compiler compliance level is 1.8 or above (but then that is needed for this version of DataNucleus 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-jdo-query.jar`, `javax.jdo.jar`

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

Using Scala

Please refer to this proof of concept project which demonstrates use of DataNucleus JDO (including Typed queries) with Scala.

Query Classes

The above preparation will mean that whenever you compile, the DataNucleus annotation processor (in `datanucleus-jdo-query.jar`) will generate a **query class** for each model class that is annotated as persistable. So what is a **query class** you ask. It is simply a mechanism for providing an intuitive API to generating queries. If we have the following model class

```
@PersistenceCapable
public class Product
{
    @PrimaryKey
    long id;
    String name;
    double value;

    ...
}
```

then the (generated) **query class** for this will be

```
public class QProduct extends org.datanucleus.api.jdo.query.PersistableExpressionImpl
<Product>
    implements PersistableExpression<Product>
{
    public static QProduct candidate(String name) {...}
    public static QProduct candidate() {...}
    public static QProduct variable(String name) {...}
    public static QProduct parameter(String name) {...}

    public NumericExpression<Long> id;
    public StringExpression name;
    public NumericExpression<Double> value;

    ...
}
```

The generated class has the name of form *Q*{className}*. Also the generated class, by default, has a public field for each persistable field/property and is of a type *XXXExpression*. These expressions allow us to give Java like syntax when defining your queries (see below). So you access your persistable members in a query as **candidate.name** for example.

As mentioned above this is the default style of query class. However you can also create it in *property* style, where you access your persistable members as **candidate.name()** for example. The benefit of this approach is that if you have 1-1, N-1 relationship fields then it only initialises the members when called, whereas in the *field* case above it has to initialise all in the constructor, so at static initialisation. You enable use of *property* mode by adding the compiler argument **-AqueryMode=PROPERTY**. All examples below use *field* mode but just add *()* after the field to see the equivalent in *property* mode

DataNucleus currently only supports generation of Q classes for persistable classes that are in their own source file, so no support for inline static persistable classes is available currently

The JDOQL Typed query mechanism only works for classes that are annotated, and not for classes that use XML metadata. This is due to the fact that it makes use of a Java *annotation processor*.

Limitations

There are some corner cases where the use of expressions and this API may require casting to allow the full range of operations for JDOQL. Some examples

- If you have a List field and call `ListExpression.get(position)` this returns an `Expression` rather than a specific `NumericExpression`, `StringExpression`, or whatever subtype. You would need to cast the result to do subsequent calls.
- If you have a Map field and call `MapExpression.get(key)` this returns an `Expression` rather than a specific `NumericExpression`, `StringExpression`, or whatever subtype. You would need to cast the result to do subsequent calls.
- If you have a Collection parameter and call `CollectionParameter.contains(fieldExpression)` then you may need to cast the `fieldExpression` to `Expression` since the `CollectionParameter` will not have adequate java generic information for the compiler to do it automatically
- If you have a Map parameter and call `MapParameter.contains(fieldExpression)` then you may need to cast the `fieldExpression` to `Expression` since the `MapParameter` will not have adequate java generic information for the compiler to do it automatically

Query API - Filtering

Let's provide a sample usage of this query API. We want to construct a query for all products with a value below a certain level, and where the name starts with "Wal". So a typical query in a JDO-enabled application

```
pm = pmf.getPersistenceManager();

JDOQLTypedQuery<Product> tq = pm.newJDOQLTypedQuery(Product.class);
QProduct cand = QProduct.candidate();
List<Product> results = tq.filter(cand.value.lt(40.00).and(cand.name.startsWith(
"Wal")))
    .executeList();
```

This equates to the single-string query

```
SELECT FROM mydomain.Product WHERE this.value < 40.0 && this.name.startsWith("Wal")
```

As you see, we create a parametrised query, and then make use of the **query class** to access the candidate, and from that make use of its fields, and the various Java methods present for the types of those fields. Note that the API is *fluent*, meaning you can chain calls easily.

Query API - Ordering

We want to order the results of the previous query by the product name, putting nulls first.

```
tq.orderBy(cand.name.asc().nullsFirst());
```

This query now equates to the single-string query

```
SELECT FROM mydomain.Product WHERE this.value < 40.0 && this.name.startsWith("Wal")
ORDER BY this.name ASCENDING NULLS FIRST
```

If you don't want to specify null positioning, simply omit the `nullsFirst()` call. Similarly to put nulls last then call `nullsLast()`.

Query API - Methods

In the above example you will have seen the use of some of the normal JDOQL methods. With the JDOQLTyped API these are available on the different types of expressions. For example, *cand.name* is a `StringExpression` and consequently it has all of the normal String methods available, just like in JDOQL and just like in Java. Similarly if we had a class `Inventory` which had a Collection of `Product`, then we could use the method **contains** on the `CollectionExpression`.

The JDOQL methods *JDOHelper.getObjectId* and *JDOHelper.getVersion* are available on `PersistableExpression`, for the object that they would be invoked on.

The JDOQL methods *Math.{xxx}* are available on `NumericExpression`, for the numeric that they would be invoked on.

GeoSpatial Object Methods

When you have fields/properties that use geospatial types, you can query these using methods in JDOQL. DataNucleus also allows use of methods using *JDOQLTypedQuery* for these types using a vendor extension.

You need to be using the DataNucleus `javax.jdo.jar` to be able to use this extension.

Firstly, a geospatial field will be mapped on to one of `GeometryExpression`, `LineStringExpression`, `PointExpression`, `PolygonExpression`, `LinearRingExpression`, `MultiLineStringExpression`, `MultiPointExpression`, or `MultiPolygonExpression`. These types have a range of methods available on them.

An example,

```
JDOQLTypedQuery<Property> tq = pm.newJDOQLTypedQuery(Property.class);
QProperty cand = QProperty.candidate();

tq.filter(cand.location.ne((Point)null).and(cand.location.getX().lt(tq.numericParamete
r("theX"))));
tq.setParameter("theX", 100.0);

List list = tq.executeList();
```

which is equivalent to the JDOQL

```
SELECT FROM mydomain.Property WHERE this.location.getX() < :theX
```

GeoSpatial Static Methods

You can also invoke static geospatial methods in JDOQLTypedQuery. You do this via use of the *GeospatialHelper*.

You need to be using the DataNucleus `javax.jdo.jar` to be able to use this extension.

```
GeospatialHelper geoHelper = tq.geospatialHelper();

GeometryExpression geomExpr = geoHelper.geometryFromText("POINT(25 45)", 4126);
```

and this expression is then available to be used in the *JDOQLTypedQuery*.

Query API - Results

Let's take the query in the above example and return the name and value of the Products only

```
JDOQLTypedQuery<Product> tq = pm.newJDOQLTypedQuery(Product.class);
QProduct cand = QProduct.candidate();
List<Object[]> results = tq.filter(cand.value.lt(40.00).and(cand.name.startsWith(
"Wal"))).orderBy(cand.name.asc())
        .result(false, cand.name, cand.value).executeResultList();
```

This equates to the single-string query

```
SELECT this.name,this.value FROM mydomain.Product WHERE this.value < 40.0 &&
this.name.startsWith("Wal") ORDER BY this.name ASCENDING
```

A further example using aggregates

```
JDOQLTypedQuery<Product> tq = pm.newJDOQLTypedQuery(Product.class);
Object results =
    tq.result(false, QProduct.candidate().value.max(), QProduct.candidate().value.
min()).executeResultUnique();
```

This equates to the single-string query

```
SELECT max(this.value), min(this.value) FROM mydomain.Product
```

If you wanted to assign an alias to a result component you do it like this

```
tq.result(false, cand.name.as("THENAME"), cand.value.as("THEVALUE"));
```

Query API - Parameters

It is important to note that JDOQLTypedQuery only accepts **named** parameters. You obtain a named parameter from the JDOQLTypedQuery, and then use it in the specification of the filter, ordering, grouping etc. Let's take the query in the above example and specify the "Wal" in a parameter.

```
JDOQLTypedQuery<Product> tq = pm.newJDOQLTypedQuery(Product.class);
QProduct cand = QProduct.candidate();
List<Product> results =
    tq.filter(cand.value.lt(40.00).and(cand.name.startsWith(tq.stringParameter
("prefix"))))
        .orderBy(cand.name.asc())
        .setParameter("prefix", "Wal").executeList();
```

This equates to the single-string query

```
SELECT FROM mydomain.Product WHERE this.value < 40.0 && this.name.startsWith(:prefix)
ORDER BY this.name ASCENDING
```

RDBMS : Parameters .v. Literals

When considering whether to embody a literal into a JDOQL Typed 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.

Query API - Variables

Let's try to find all Inventory objects containing a Product with a particular name. This means we need to use a variable. Just like with a parameter, we obtain a *variable* from the Q class.

```
JDOQLTypedQuery<Inventory> tq = pm.newJDOQLTypedQuery(Inventory.class);
QProduct var = QProduct.variable("var");
QInventory cand = QInventory.candidate();
List<Inventory> results = tq.filter(cand.products.contains(var).and(var.name
.startsWith("Wal"))).executeList();
```

This equates to the single-string query

```
SELECT FROM mydomain.Inventory WHERE this.products.contains(var) && var.name
.startsWith("Wal")
```

Query API - If-Then-Else

Let's make use of an IF-THEN-ELSE expression to return the products based on whether they are "domestic" or "international" (in our case its just based on the "id")

```
JDOQLTypedQuery<Product> tq = pm.newJDOQLTypedQuery(Product.class);
QProduct cand = QProduct.candidate();
IfThenElseExpression<String> ifElseExpr = tq.ifThenElse(String.class, cand.id.lt(
1000), "Domestic", "International");
tq.result(false, ifElseExpr);
List<String> results = tq.executeResultList();
```

This equates to the single-string query

```
SELECT IF (this.id < 1000) "Domestic" ELSE "International" FROM mydomain.Product
```

Query API - Subqueries

Let's try to find all Products that have a value below the average of all Products. This means we need to use a subquery

```
JDOQLTypedQuery<Product> tq = pm.newJDOQLTypedQuery(Product.class);
QProduct cand = QProduct.candidate();
TypesafeSubquery<Product> tqsub = tq.subquery(Product.class, "p");
QProduct candsub = QProduct.candidate("p");
List<Product> results = tq.filter(cand.value.lt(tqsub.selectUnique(candsub.value.
avg()))).executeList();
```

Note that where we want to refer to the candidate of the subquery, we specify the alias ("p") explicitly. This equates to the single-string query

```
SELECT FROM mydomain.Product WHERE this.value < (SELECT AVG(p.value) FROM
mydomain.Product p)
```

When you are using a subquery and want to refer to the candidate (or field thereof) of the outer query in the subquery then you would use `cand` in the above example (or a field of it as required).

Query API - Candidates

If you don't want to query instances in the datastore but instead query a collection of candidate instances, you can do this by setting the candidates, like this

```
JDOQLTypedQuery<Product> tq = pm.newJDOQLTypedQuery(Product.class);
QProduct cand = QProduct.candidate();
List<Product> results = tq.filter(cand.value.lt(40.00)).setCandidates(myCandidates
).executeList();
```

This will process the query in-memory.

SQL

As we have described earlier, JDO allows access to many query languages to give the user full flexibility over what they utilise. Sometimes an object-based query language (such as JDOQL) is not considered suitable, maybe due to the lack of familiarity of the application developer with such a query language. In the case where you are using an RDBMS it is sometimes desirable to query using **SQL**. JDO standardises this as a valid query mechanism, and DataNucleus supports this.

Please be aware that the SQL query that you invoke has to be valid for your RDBMS, and that the SQL syntax differs across almost all RDBMS.

To utilise **SQL** syntax in queries, you create a Query as follows

```
Query q = pm.newQuery("javax.jdo.query.SQL", the_sql_query);
```

You have several forms of SQL queries, depending on what form of output you require.

- **No candidate class and no result class** - the result will be a List of Objects (when there is a single column in the query), or a List of Object[]s (when there are multiple columns in the query)
- **Candidate class specified, no result class** - the result will be a List of candidate class objects, or will be a single candidate class object (when you have specified "unique"). The columns of the querys result set are matched up to the fields of the candidate class by name. You need to select a minimum of the PK columns in the SQL statement.
- **No candidate class, result class specified** - the result will be a List of result class objects, or will be a single result class object (when you have specified "unique"). Your result class has to abide by the rules of JDO result classes (see Result Class specification) - this typically means either providing public fields matching the columns of the result, or providing setters/getters for the columns of the result.
- **Candidate class and result class specified** - the result will be a List of result class objects, or will be a single result class object (when you have specified "unique"). The result class has to abide by the rules of JDO result classes (see Result Class specification).

Setting candidate class

If you want to return instances of persistable types, then you can set the candidate class.

```
Query query = pm.newQuery("javax.jdo.query.SQL", "SELECT MY_ID, MY_NAME FROM
MYTABLE");
query.setClass(MyClass.class);
List<MyClass> results = query.executeList();
```

Unique results

If you know that there will only be a single row returned from the SQL query then you can set the query as *unique*. Note that the query will return null if the SQL has no results.

Sometimes you know that the query can only every return 0 or 1 objects. In this case you can simplify your job by adding

```
// Using traditional JDO Query API
Query query = pm.newQuery("javax.jdo.query.SQL", "SELECT MY_ID, MY_NAME FROM
MYTABLE");
query.setClass(MyClass.class);
query.setUnique(true);
MyClass obj = (MyClass) query.execute();

// Using JDO3.2 Query API
Query query = pm.newQuery("javax.jdo.query.SQL", "SELECT MY_ID, MY_NAME FROM
MYTABLE");
query.setClass(MyClass.class);
MyClass obj = query.executeUnique();
```

Defining a result type

If you want to dump each row of the SQL query results into an object of a particular type then you can set the result class.

```
// Using traditional JDO Query API
Query query = pm.newQuery("javax.jdo.query.SQL", "SELECT MY_ID, MY_NAME FROM
MYTABLE");
query.setResultClass(MyResultClass.class);
List<MyResultClass> results = (List<MyResultClass>) query.execute();


// Using JDO3.2 Query API
Query query = pm.newQuery("javax.jdo.query.SQL", "SELECT MY_ID, MY_NAME FROM
MYTABLE");
List<MyResultClass> results = query.executeResultList(MyResultClass.class);
```

The *Result Class* has to meet certain requirements. These are

- Can be one of Integer, Long, Short, Float, Double, Character, Byte, Boolean, String, java.math.BigInteger, java.math.BigDecimal, java.util.Date, java.sql.Date, java.sql.Time, java.sql.Timestamp, or Object[]
- Can be a user defined class, that has either a constructor taking arguments of the same type as those returned by the query (in the same order), or has a public put(Object, Object) method, or public setXXX() methods, or public fields.

For example, if we are returning two columns like above, an *int* and a *String* then we define our result class like this

```
public class MyResultClass
{
    protected int id = 0;
    protected String name = null;

    public MyResultClass(int id, String name)
    {
        this.id = id;
        this.name = name;
    }

    ...
}
```

So here we have a result class using the constructor arguments. We could equally have provided a class with public fields instead, or provided *setXXX* methods, or just provide a *put* method. They all work in the same way.

SQL Syntax Checks

When an SQL query is a SELECT, and is returning instances of an persistable class, 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.query.sql.syntaxChecks** to *false*. Similarly you can turn them off on a query-by-query basis by setting the query hint **datanucleus.query.sql.syntaxChecks** to *false*.

Inserting/Updating/Deleting

In strict JDO all SQL queries must begin "SELECT ...", and consequently it is not possible to execute queries that change data. In DataNucleus we have an extension that allows this to be overridden; to enable this you should specify the persistence property **datanucleus.query.sql.allowAll** as *true*, and thereafter you just invoke your statements like this

```
Query q = pm.newQuery("javax.jdo.query.SQL", "UPDATE MY_TABLE SET MY_COLUMN = ? WHERE
MY_ID = ?");
```

you then pass any parameters in as normal for an SQL query. If your query starts with "SELECT" then it is invoked using *preparedStatement.executeQuery(...)*. If your query starts with "UPDATE", "INSERT", "MERGE", "DELETE" it is treated as a bulk update/delete query and is invoked using *preparedStatement.executeUpdate(...)*. All other statements will be invoked using *preparedStatement.execute(...)* and true returned.

If your statement really needs to be executed differently to these basic rules then you should look at contributing support for those statements to DataNucleus.

Parameters

In JDO SQL queries can have parameters but must be *positional*. This means that you do as follows

```
Query q = pm.newQuery("javax.jdo.query.SQL", "SELECT col1, col2 FROM MYTABLE WHERE
col3 = ? AND col4 = ? and col5 = ?");
List results = q.setParameters(val1, val2, val3).executeList();
```

So we used traditional JDBC form of parametrisation, using "?".

DataNucleus also supports two further variations. The first is called *numbered* parameters where we assign numbers to them, so the previous example could have been written like this

```
Query q = pm.newQuery("javax.jdo.query.SQL", "SELECT col1, col2 FROM MYTABLE WHERE
col3 = ?1 AND col4 = ?2 and col5 = ?1");
List results = q.setParameters(val1, val2).executeList();
```

so we can reuse parameters in this variation. The second variation is called *named* parameters where we assign names to them, and so the example can be further rewritten like this

```
Query q = pm.newQuery("javax.jdo.query.SQL", "SELECT col1, col2 FROM MYTABLE WHERE
col3 = :firstVal AND col4 = :secondVal and col5 = :firstVal");
Map params = new HashMap();
params.put("firstVal", val1);
params.put("secondVal", val1);
List results = q.setNamedParameters(params).executeList();
```

Example 1 - Using SQL aggregate functions, without candidate class

Here's an example for getting the size of a table without a candidate class.

```
Query query = pm.newQuery("javax.jdo.query.SQL", "SELECT count(*) FROM MYTABLE");
List results = query.executeList();
Integer tableSize = (Integer) result.iterator().next();
```

Here's an example for getting the maximum and miminum of a parameter without a candidate class.

```
Query query = pm.newQuery("javax.jdo.query.SQL", "SELECT max(PARAM1), min(PARAM1) FROM
MYTABLE");
List results = query.executeList();
Object[] measures = (Object[])result.iterator().next();
Double maximum = (Double)measures[0];
Double minimum = (Double)measures[1];
```

Example 2 - Using SQL aggregate functions, with result class

Here's an example for getting the size of a table with a result class. So we have a result class of

```
public class TableStatistics
{
    private int total;

    public setTotal(int total);
}
```

So we define our query to populate this class

```
Query query = pm.newQuery("javax.jdo.query.SQL", "SELECT count(*) AS total FROM
MYTABLE");
List<TableStatistics> results = query.executeResultList(TableStatistics.class);
TableStatistics tableStats = result.iterator().next();
```

Each row of the results is of the type of our result class. Since our query is for an aggregate, there is actually only 1 row.

Example 3 - Retrieval using candidate class

When we want to retrieve objects of a particular persistable class we specify the candidate class. Here we need to select, as a minimum, the identity columns for the class.

```
Query query = pm.newQuery("javax.jdo.query.SQL", "SELECT MY_ID, MY_NAME FROM
MYTABLE");
query.setClass(MyClass.class);
List<MyClass> results = query.executeList();
Iterator resultsIter = results.iterator();
while (resultsIter.hasNext())
{
    MyClass obj = resultsIter.next();
}
```

```
class MyClass
{
    String name;
    ...
}
```

```
<package name="mydomain.samples.sql">
    <class name="MyClass" identity-type="datastore" table="MYTABLE">
        <datastore-identity strategy="identity">
            <column name="MY_ID"/>
        </datastore-identity>
        <field name="name" persistence-modifier="persistent">
            <column name="MY_NAME"/>
        </field>
    </class>
</package>
```

Example 4 - Using parameters, without candidate class

Here's an example for getting the number of people with a particular email address. You simply add a "?" for all parameters that are passed in, and these are substituted at execution time.

```
Query query = pm.newQuery("javax.jdo.query.SQL", "SELECT count(*) FROM PERSON WHERE
EMAIL_ADDRESS = ?");
List results = query.setParameters("nobody@datanucleus.org").executeList();
Integer tableSize = (Integer) result.iterator().next();
```

Example 5 - Named Query

While "named" queries were introduced primarily for JDOQL queries, we can define "named" queries for SQL also. So let's take a *Product* class, and we want to define a query for all products that are "sold out". We firstly add this to our MetaData

```
<package name="mydomain.samples.store">
    <class name="Product" identity-type="datastore" table="PRODUCT">
        <datastore-identity strategy="identity">
            <column name="PRODUCT_ID"/>
        </datastore-identity>
        <field name="name" persistence-modifier="persistent">
            <column name="NAME"/>
        </field>
        <field name="status" persistence-modifier="persistent">
            <column name="STATUS"/>
        </field>

        <query name="SoldOut" language="javax.jdo.query.SQL">
            SELECT PRODUCT_ID FROM PRODUCT WHERE STATUS == "Sold Out"
        </query>
    </class>
</package>
```

And then in our application code we utilise the query

```
Query q = pm.newNamedQuery(Product.class, "SoldOut");
List<Product> results = q.executeList();
```

Cassandra CQL

As we have described earlier, JDO allows access to many query languages to give the user full flexibility over what they utilise. Sometimes an object-based query language (such as JDOQL) is not considered suitable, maybe due to the lack of familiarity of the application developer with such a query language. In the case where you are using Cassandra it is sometimes desirable to query using **CQL**. JDO provides a mechanism to use this as a valid query mechanism, and DataNucleus supports this.

To utilise **CQL** syntax in queries with Cassandra datastores, you create a Query as follows

```
Query q = pm.newQuery("CQL", "SELECT * FROM schema1.Employee");
// Fetch 10 Employee rows at a time
query.getFetchPlan().setFetchSize(10);
query.setResultClass(Employee.class);
List<Employee> results = (List)q.execute();
```

You can also query results as List<Object[]> without specifying a specific result type as shown below.

```
// Find all employees
PersistenceManager persistenceManager = pmf.getPersistenceManager();
Query q = pm.newQuery("CQL", "SELECT * FROM schema1.Employee");
// Fetch all Employee rows as Object[] at a time.
query.getFetchPlan().setFetchSize(-1);
List<Object[]> results = (List)q.execute();
```

So we are utilising the JDO API to generate a query and passing in the Cassandra "CQL".

JPQL

As we have described earlier, JDO allows access to many query languages to give the user full flexibility over what they utilise. It may be that the developers of your project are familiar with the JPA query language *JPQL*. DataNucleus allows full support for this syntax.

You would create and execute a JPQL query using the JDO API like this

```
Query q = pm.newQuery("JPQL", "SELECT p FROM Person p WHERE p.lastName = 'Jones'");
List results = (List)q.execute();
```

This finds all "Person" objects with surname of "Jones". You specify all details in the query.

You can find full details of the JPQL syntax in the JPQL Query Guide for JPA

Since you are using the JDO API here, there may be some parts of JPA mapping metadata that is not available for use with JDO, so we note some known differences below.

Entity Name

In the example shown you note that we did not specify the full class name. We used *Person p* and thereafter could refer to *p* as the alias. The *Person* is called the **entity name** and in JPA MetaData this can be defined against each class in its definition. With JDO we don't have this MetaData attribute so we simply define the **entity name** as *the name of the class omitting the package name*. So *mydomain.samples.Person* will have an entity name of *Person*.

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 1-N/M-N fields where possible. 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. For RDBMS, we respect this in some specific situations only.

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

Stored Procedures

applicable to RDBMS.

JDO doesn't include explicit support for stored procedures. However DataNucleus provides two options for allowing use of stored procedures.

Using DataNucleus Stored Procedure API

Obviously JDO allows potentially any "query language" to be invoked using its API. We can do the following

```
Query q = pm.newQuery("STOREDPROC", "MY_TEST_SP_1");
```

Now on its own this will simply invoke the define stored procedure (*MY_TEST_SP_1*) in the datastore. Obviously we want more control than that, so this is where you use DataNucleus specifics. Let's start by accessing the internal stored procedure query

```
import org.datanucleus.api.jdo.JDOQuery;
import org.datanucleus.store.rdbms.query.StoredProcedureQuery;
...
StoredProcedureQuery spq = (StoredProcedureQuery)((JDOQuery)q).getInternalQuery());
```

You should familiarise yourself with the StoredProcedureQuery 

API. Bear in mind that this extends the normal *Query* API, and so you set parameters using that.

Now we can control things like parameters, and what is returned from the stored procedure query. Let's start by registering any parameters (IN, OUT, or INOUT) for our stored proc. In our example we use named parameters, but you can also use positional parameters.

```
spq.registerParameter("PARAM1", String.class, StoredProcQueryParameterMode.IN);
spq.registerParameter("PARAM2", Integer.class, StoredProcQueryParameterMode.OUT);
```

Simple execution is like this (where you omit the paramValueMap if you have no input parameters).

```
boolean hasResultSet = spq.executeWithMap(paramValueMap);
```

That method returns whether a result set is returned from the stored procedure (some return results, but some return an update count, and/or output parameters). If we are expecting a result set we then do

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

and if we are expecting output parameter values then we get them using the API too. Note again that you can also access via position rather than name.

```
Object val = spq.getOutputParameterValue("PARAM2");
```

That summarises our stored procedure API. It also allows things like multiple result sets for a stored procedure, all using the *StoredProcedureQuery* API.

Using JDO SQL Query API to invoke stored procedures

In JDO all SQL queries must begin "SELECT ...", and consequently it is not possible to execute stored procedures by default. In DataNucleus we have an extension that allows this to be overridden, to call stored procedures.

This is strongly discouraged now that we provide the mechanism above

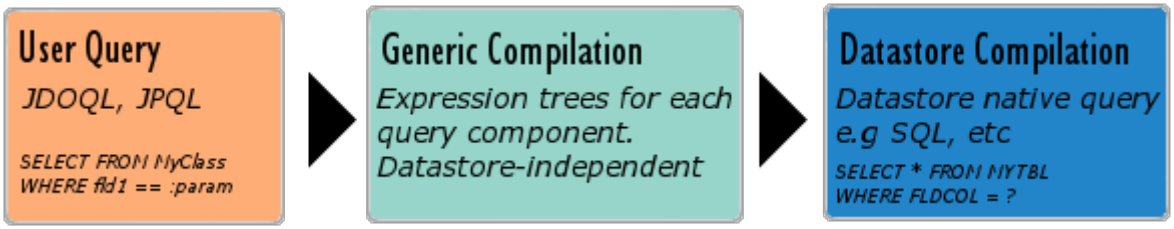
To enable this you should specify the persistence property **datanucleus.query.sql.allowAll** as *true* when creating the PMF. Thereafter you can invoke your stored procedures like this

```
Query q = pm.newQuery("javax.jdo.query.SQL", "EXECUTE sp_who");
((org.datanucleus.api.jdo.JDOQuery)q).getInternalQuery().setType(org.datanucleus.store
.query.Query.SELECT);
```

Where "sp_who" is the stored procedure being invoked. The syntax of calling a stored procedure differs across RDBMS, some require "CALL ..." and some "EXECUTE ..."; Go consult your manual. Clearly the same rules will apply regarding the results of the stored procedure and mapping them to any result class.

Query Cache

Extension



JDO 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

Query Result caching is, by default, turned **OFF**, since we have to select what is appropriate for the vast majority of usages. You can turn caching on/off (default = off) by using the persistence property **datanucleus.query.results.cached** set to (true|false), and likely better on a query-by-query basis by specifying the query extension **datanucleus.query.results.cached** as (true|false).

This cache type 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.

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.jdo.JDOQueryCache;
import org.datanucleus.api.jdo.JDOPersistenceanagerFactory;

...
JDOQueryCache cache = ((JDOPersistenceManagerFactory)pmf).getQueryCache();

cache.evict(query);
```

which evicts the results of the specific query. The JDOQueryCache Javadoc has more options available should you need them - see its API.