OJB Queries

by Jakob Braeuchi

1. Introduction

This tutorial describes the use of the different queries mechanisms. The sample code shown here is taken mainly from JUnit test classes. The junit test source can be found under `[db-ojb]/src/test` in the source distribution.

2. Query by Criteria

In this section you will learn how to use the query by criteria. The classes are located in the package `org.apache.ojb.broker.query`. Using query by criteria you can either query for whole objects (ie. person) or you can use report queries returning row data.

A query consists mainly of the following parts:

1. the class of the objects to be retrieved
2. a list of criteria
3. a DISTINCT flag
4. additional ORDER BY and GROUP BY

OJB offers a QueryFactory to create a new Query. Although the constructors of the query classes are public using the QueryFactory is the preferred way to create a new query.

```
Query q = QueryFactory.newQuery(Person.class, crit);
```

To create a DISTINCT-Query, simply add **true** as third parameter.

```
Query q = QueryFactory.newQuery(Person.class, crit, true);
```

Each criterion stands for a column in the SQL-WHERE-clause.

```
Criteria crit = new Criteria();
crit.addEqualTo("upper(firstname)", "TOM");
crit.addEqualTo("lastname", "hanks");
Query q = QueryFactory.newQuery(Person.class, crit);
```

This query will generate an SQL statement like this:

```
SELECT ... FROM PERSON WHERE upper(FIRSTNAME) = "TOM" AND LASTNAME = "hanks";
```

OJB supports **functions** in field criteria ie. upper(firstname). When converting a field name to a database column name, the function is added to the generated sql. OJB does not and can not verify the correctness of the specified function, an illegal function will produce an SqlException.

2.1. Query Criteria

OJB provides selection criteria for almost any SQL-comparator. In most cases you do not have to deal directly with the implementing classes like *EqualToCriteria*. The *Criteria* class provides factory methods for the appropriate classes. There are four kinds of factory methods:

- create criteria to compare a field to a value: ie. addEqualTo("firstname", "tom");
- create criteria to compare a field to another field: ie. addEqualToField("firstname", "other_field");

- create criteria to check null value: ie. addIsNull("firstname");
- create a raw sql criteria: ie: addSql("REVERSE(name) like 're%'");

The following list shows some of the factory methods to compare a field to a value:

- addEqualTo
- addLike
- addGreaterOrEqualThan
- addGreaterThan
- addLike
- addBetween , this methods has two value parameters
- addIn , this method uses a Collection as value parameter
- and of course there negative forms

This list shows some factory methods to compare a field to another field, all those methods end on ...field:

- addEqualToField
- addGreaterThanField
- and of course there negative forms

2.1.1. in / not in

Some databases limit the number of parameters in an IN-statement.
If the limit is reached OJB will split up the IN-Statement into multiple Statements, the limit is set to 3 for the following sample:

```
SELECT ... FROM Artikel A0 WHERE A0.Kategorie_Nr IN ( ? , ? , ? )
OR A0.Kategorie_Nr IN ( ? , ? ) ORDER BY 7 DESC
```

The IN-limit for prefetch can be defined in OJB.properties:

```
...
# The SqlInLimit entry limits the number of values in IN-sql
# statement, -1 for no limits. This hint is used in Criteria.
SqlInLimit=200
...
```

2.1.2. and / or

All selection criteria added to a criteria set using the above factory methods will be **AND**ed in the WHERE-clause. To get an **OR** combination two criteria sets are needed. These sets are combined using addOrCriteria:

```
Criteria crit1 = new Criteria();
crit1.addLike("firstname", "%o%");
crit1.addLike("lastname", "%m%");
Criteria crit2 = new Criteria();
crit2.addEqualTo("firstname", "hank");

crit1.addOrCriteria(crit2);
Query q = QueryFactory.newQuery(Person.class, crit1);

Collection results = broker.getCollectionByQuery(q);
```

This query will generate an SQL statement like this:

```
SELECT ... WHERE (FIRSTNAME LIKE "%o%") AND LASTNAME
LIKE "%m%" OR FIRSTNAME = "hank"
```

2.1.3. negating the criteria

A criteria can be negated to obtain **NOT** in the WHERE-clause:

```
Criteria crit1 = new Criteria();
crit1.addLike("firstname", "%o%");
crit1.addLike("lastname", "%m%");
crit1.setNegative(true);

Collection results = broker.getCollectionByQuery(q);
```

This query will generate an SQL statement like this:

```
SELECT ... WHERE NOT (FIRSTNAME LIKE "%o%" AND LASTNAME LIKE "%m%")
```

2.2. ordering and grouping

The following methods of QueryByCriteria are used for ordering and grouping:

- addOrderByAscending(String anAttributeName);
- addOrderByDescending(String anAttributeName);
- addGroupBy(String anAttributeName); this method is used for report queries

You can of course have multiple order by and group by clauses, simply repeat the addOrderBy.

```
crit = new Criteria();
query = new QueryByCriteria(Person.class, crit);
query.addOrderByDescending("id");
query.addOrderByAscending("lastname");
broker.getCollectionByQuery(query);
```

The code snippet will query all Persons and order them by **attribute** "id" descending and "lastname" ascending. The query will produce the following SQL-statement using column numbers in the ORDER BY clause:

```
SELECT A0.ID,A0.FIRSTNAME,A0.LASTNAME FROM
PERSON A0 ORDER BY 1 DESC, 3
```

When you use the **column** name "LASTNAME" instead of the **attribute** name "lastname" (query.addOrderBy("LASTNAME");), an additional column named "LASTNAME" without alias will be added.

```
SELECT A0.ID,A0.FIRSTNAME,A0.LASTNAME,LASTNAME FROM
PERSON A0 ORDER BY 1 DESC,4
```

If there are multiple tables with a column "LASTNAME" the SQL-Statement will produce an error, so it's better to always use attribute names.

2.3. subqueries

Subqueries can be used instead of values in selection criteria. The subquery should thus be a ReportQuery.
The following example queries all articles having a price greator or equal than the average price of articles named 'A%':

```
ReportQueryByCriteria subQuery;
Criteria subCrit = new Criteria();
Criteria crit = new Criteria();

subCrit.addLike("articleName", "A%");
subQuery = QueryFactory.newReportQuery(Article.class, subCrit);
subQuery.setAttributes(new String[] { "avg(price)" });

crit.addGreaterOrEqualThan("price", subQuery);
Query q = QueryFactory.newQuery(Article.class, crit);

Collection results = broker.getCollectionByQuery(q);
```

It's also possible to build a subquery with attributes referencing the enclosing query. These attributes have to use a special prefix **Criteria.PARENT_QUERY_PREFIX**.
The following example queries all product groups having more than 10 articles:

```
ReportQueryByCriteria subQuery;
Criteria subCrit = new Criteria();
Criteria crit = new Criteria();

subCrit.addEqualToField("productGroupId", Criteria.PARENT_QUERY_PREFIX + "groupId");
subQuery = QueryFactory.newReportQuery(Article.class, subCrit);
subQuery.setAttributes(new String[] { "count(productGroupId)" });

crit.addGreaterThan(subQuery, "10"); // MORE than 10 articles
crit.addLessThan("groupId", new Integer(987654));
Query q = QueryFactory.newQuery(ProductGroup.class, crit);

Collection results = broker.getCollectionByQuery(q);
```

2.4. joins

Joins resulting from **path expressions** ("relationship.attribute") in criteria are automatically handled by OJB. Path expressions are supported for all relationships 1:1, 1:n and m:n (decomposed and non-decomposed) and can be nested.

The following sample looks for all articles belonging to the product group "Liquors". Article and product group are linked by the relationship "productGroup" in class Article:

```
<!-- Definitions for org.apache.ojb.ojb.broker.Article -->
   <class-descriptor
         class="org.apache.ojb.broker.Article"
         proxy="dynamic"
         table="Artikel"
   >
      ...
      <reference-descriptor
         name="productGroup"
         class-ref="org.apache.ojb.broker.ProductGroup"
      >
         <foreignkey field-ref="productGroupId"/>
      </reference-descriptor>
   </class-descriptor>

   <class-descriptor
         class="org.apache.ojb.broker.ProductGroup"
         proxy="org.apache.ojb.broker.ProductGroupProxy"
         table="Kategorien"
   >
      ...
      <field-descriptor
         name="groupName"
         column="KategorieName"
         jdbc-type="VARCHAR"
      />
      ...
   </class-descriptor>
```

The path expression includes the 1:1 relationship "productGroup" and the attribute "groupName":

```
Criteria crit = new Criteria();
crit.addEqualTo("productGroup.groupName", "Liquors");
Query q = QueryFactory.newQuery(Article.class, crit);

Collection results = broker.getCollectionByQuery(q);
```

If path expressions refer to a class having **extents**, the tables of the extent classes participate in the JOIN and the criteria is ORed. The shown sample queries all ProductGroups having an Article named 'F%'. The path expression 'allArticlesInGroup' refers to the class Articles which has two extents: Books and CDs.

```
Criteria crit = new Criteria();
crit.addLike("allArticlesInGroup.articleName", "F%");
```

```
QueryByCriteria q = QueryFactory.newQuery(ProductGroup.class, crit, true);

Collection results = broker.getCollectionByQuery(q);
```

This sample produces the following SQL:

```
SELECT DISTINCT A0.KategorieName,A0.Kategorie_Nr,A0.Beschreibung
FROM Kategorien A0
INNER JOIN Artikel A1 ON A0.Kategorie_Nr=A1.Kategorie_Nr
LEFT OUTER JOIN BOOKS A1E0 ON A0.Kategorie_Nr=A1E0.Kategorie_Nr
LEFT OUTER JOIN CDS A1E1 ON A0.Kategorie_Nr=A1E1.Kategorie_Nr
WHERE A1.Artikelname LIKE  'F%'  OR
A1E0.Artikelname LIKE  'F%'  OR
A1E1.Artikelname LIKE  'F%'
```

OJB tries to do it's best to automatically use **outer** joins where needed. This is currently the case for classes having extents and ORed criteria. But you can force the SQLGenerator to use outer joins where you find it useful.
This is done by the method *QueryByCriteria#setPathOuterJoin(String)*.

```
ReportQueryByCriteria query;
Criteria crit;
Iterator result1, result2;

crit = new Criteria();

query = new ReportQueryByCriteria(Person.class, crit);
query.setAttributes(new String[] { "id", "name", "vorname", "sum(konti.saldo)" });
query.addGroupBy(new String[]{ "id", "name", "vorname" });

result1 = broker.getReportQueryIteratorByQuery(query);

query.setPathOuterJoin("konti");
result2 = broker.getReportQueryIteratorByQuery(query);
```

The first query will use an inner join for relationship "konti", the second an outer join.

2.5. user defined alias

This feature allows to have **multiple** aliases for the same table. The standard behaviour of OJB is to build one alias for one relationship.

Suppose you have two classes Issue and Keyword and there is a 1:N relationship between them. Now you want to retrieve Issues by querying on Keywords. Suppose you want to retrieve all Issues with keywords 'JOIN' and 'ALIAS'. If these values are stored in the attribute 'value' of Keyword, OJB generates a query that contains " A1.value = 'JOIN' AND A1.value = 'ALIAS' " in the where-clause. Obviously, this will not work, no hits will occur because A1.value can not have more then 1 value at the time !

For the examples below, suppose you have the following classes (pseudo-code):

```
class Container
    int id
    Collection allAbstractAttributes

class AbstractAttribute
    int id
    inf ref_id
    String name
    String value
    Collection allAbstractAttributes
```

OJB maps these classes to separate tables where it maps allAbstractAttributes using a collectiondescriptor to AbstractAttribute using ref_id as inverse foreignkey on Container for the collection descriptor.
For demo purposes : AbstractAttribute also has a collection of abstract attributes.

```
Criteria crit1 = new Criteria();
crit1.setAlias("company");                  // set an alias
crit1.addEqualTo("allAbstractAttributes.name", "companyName");
crit1.addEqualTo("allAbstractAttributes.value", "iBanx");

Criteria crit2 = new Criteria();
crit2.setAlias("contact");                  // set an alias
crit2.addEqualTo("allAbstractAttributes.name", "contactPerson");
crit2.addLike("allAbstractAttributes.value", "janssen");

Criteria crit3 = new Criteria();
crit3.addEqualTo("allAbstractAttributes.name", "size");
crit3.addGreaterThan("allAbstractAttributes.value", new Integer(500));

crit1.addAndCriteria(crit2);
crit1.addAndCriteria(crit3);

q = QueryFactory.newQuery(Container.class, crit1);
q.addOrderBy("company.value");     // user alias
```

The generated query will be as follows. Note that the alias name 'company' does not show up in the SQL.

```
SELECT DISTINCT A0.ID, A1.VALUE
FROM CONTAINER A0 INNER JOIN ABSTRACT_ATTRIBUTE A1
     ON A0.ID=A1.REF_ID
     INNER JOIN ABSTRACT_ATTRIBUTE A2
     ON A0.ID=A2.REF_ID
     INNER JOIN ABSTRACT_ATTRIBUTE A3
     ON A0.ID=A3.REF_ID
WHERE (( A0.NAME =  'companyName' ) AND  (A0.VALUE =  'iBanx' )) AND
      (( A1.NAME =  'contactPerson' ) AND  (A1.VALUE LIKE '%janssen%' )) AND
      (( A2.NAME =  'size' ) AND  (A2.VALUE =  '500' ))
ORDER BY 2
```

The next example uses a report query.

```
Criteria crit1 = new Criteria();
crit1.setAlias("ALIAS1");
crit1.addEqualTo("allAbstractAttributes.allAbstractAttributes.name", "xxxx");
crit1.addEqualTo("allAbstractAttributes.allAbstractAttributes.value", "hello");

Criteria crit2 = new Criteria();
crit2.setAlias("ALIAS2");
crit2.addEqualTo("allAbstractAttributes.name", "yyyy");
crit2.addLike("allAbstractAttributes.value", "");

crit1.addAndCriteria(crit2);

q = QueryFactory.newReportQuery(Container.class, crit1);

String[] cols = { id, "ALIAS2.name", "ALIAS2.name", "ALIAS1.name", "ALIAS1.name" };
q.setAttributes(cls);
```

The generated query will be:

```
SELECT DISTINCT A0.ID, A1.NAME, A1.VALUE, A2.NAME, A2.VALUE
FROM CONTAINER A0 INNER JOIN ABSTRACT_ATTRIBUTE A1
     ON A0.ID=A1.REF_ID
     INNER JOIN ABSTRACT_ATTRIBUTE A2
     ON A1.ID=A2.REF_ID
WHERE (( A2.NAME =  'xxxx' ) AND  (A2.VALUE =  'hello' )) AND
      (( A1.NAME =  'yyyy' ) AND  (A2.VALUE LIKE '%%' )) AND
ORDER BY 2
```

Note:

When you define an alias for a criteria, you have to make sure that *all* attributes used in this criteria belong to the *same* class. If you break this rule OJB will probably use a wrong ClassDescriptor to resolve your attributes !

2.6. class hints

This feature allows the user to specify which class of an extent to use for a path-segment. The standard behaviour of OJB is to use the base class of an extent when it resolves a path-segment.

In the following sample the path **allArticlesInGroup** points to class Article, this is defined in the repository.xml. Assume we are only interested in ProductGroups containing CdArticles performed by Eric Clapton or Books authored by Eric Clapton, a class hint can be defined for the path. This hint is defined by:
Criteria#**addPathClass**("allArticlesInGroup", CdArticle.class);

```
//
// find a ProductGroup with a CD or a book by a particular artist
//
String artistName = new String("Eric Clapton");
crit1 = new Criteria();
crit1.addEqualTo("allArticlesInGroup.musicians", artistName);
crit1.addPathClass("allArticlesInGroup", CdArticle.class);

crit2 = new Criteria();
crit2.addEqualTo("allArticlesInGroup.author", artistName);
crit2.addPathClass("allArticlesInGroup", BookArticle.class);

crit1.addOrCriteria(crit2);

query = new QueryByCriteria(ProductGroup.class, crit1);
broker.getObjectByQuery(query);
```

Note:

This feature is also available in class QueryByCriteria but using it on Criteria-level provides additional flexibility. QueryByCriteria#addPathClass is only useful for ReportQueries to limit the class of the selected columns.

2.7. prefetched relationships

This feature can help to minimize the number of queries when reading objects with relationships. In our Testcases we have ProductGroups with a one to many relationship to Articles. When reading the ProductGroups one query is executed to get the ProductGroups and for **each** ProductGroup another query is executed to retrieve the Articles.

With prefetched relationships OJB tries to read all Articles belonging to the ProductGroups in **one** query. See further down why one query is not always possible.

```
Criteria crit = new Criteria();
crit.addLessOrEqualThan("groupId", new Integer(5));

QueryByCriteria q = QueryFactory.newQuery(ProductGroup.class, crit);
q.addOrderByDescending("groupId");
q.addPrefetchedRelationship("allArticlesInGroup");

Collection results = broker.getCollectionByQuery(q);
```

The first query reads all matching ProductGroups:

```
SELECT ... FROM Kategorien A0 WHERE
A0.Kategorie_Nr <= ? ORDER BY 3 DESC
```

The second query retrieves Articles belonging to the ProductGroups read by the first query:

```
SELECT ... FROM Artikel A0 WHERE A0.Kategorie_Nr
IN ( ? , ? , ? , ? , ? ) ORDER BY 7 DESC
```

After reading all Articles they are associated with their ProductGroup.

Note:
This function is not yet supported for relationships using Arrays.

Some databases limit the number of parameters in an IN-statement. If the limit is reached OJB will split up the second query into multiple queries, the limit is set to 3 for the following sample:

```
SELECT ... FROM Artikel A0 WHERE A0.Kategorie_Nr
IN ( ? , ? , ? ) ORDER BY 7 DESC
SELECT ... FROM Artikel A0 WHERE A0.Kategorie_Nr
IN ( ? , ? ) ORDER BY 7 DESC
```

The IN-limit for prefetch can be defined in OJB.properties SqlInLimit.

2.8. querying for objects

OJB queries return **complete** objects, that means all instance variables are filled and all 'auto-retrieve' relationships are loaded. Currently there's no way to retrieve partially loaded objects (ie. only first- and lastname of a person).

More info about manipulation of metadata setting here.

2.9. Report Queries

Report queries are used to retrieve row data, not 'real' business objects. A row is an array of Object. With these queries you can define what attributes of an object you want to have in the row. The attribute names may also contain path expressions like 'owner.address.street'. To define the attributes use ReportQuery **#setAttributes(String[] attributes)**.

The following ReportQuery retrieves the name of the ProductGroup, the name of the Article etc. for all Articles named like "C%":

```
Criteria crit = new Criteria();
Collection results = new Vector();
crit.addLike("articleName", "C%");
ReportQueryByCriteria q = QueryFactory.newReportQuery(Article.class, crit);
q.setAttributes(new String[] { "productGroup.groupName","articleId", "articleName", "price" });

Iterator iter = broker.getReportQueryIteratorByQuery(q);
```

The ReportQuery returns an Iterator over a Collection of Object[4] ([String, Integer, String, Double]).

2.9.1. Limitations of Report Queries

ReportQueries should not be used with columns referencing classes with extents. Assume we want to select all ProductGroups and summarize the amount and prize of items in stock per group. The class Article referenced by **allArticlesInGroup** has the extents Books and CDs.

```
Criteria crit = new Criteria();
Collection results = new Vector();
ReportQueryByCriteria q = QueryFactory.newReportQuery(ProductGroup.class, crit);
q.setAttributes(new String[] { "groupName", "sum(allArticlesInGroup.stock)",
"sum(allArticlesInGroup.price)" });
q.addGroupBy("groupName");

Iterator iter = broker.getReportQueryIteratorByQuery(q);
```

The ReportQuery looks quite reasonable, but it will produce an SQL not suitable for the task:

```
SELECT A0.KategorieName,sum(A1.Lagerbestand),sum(A1.Einzelpreis)
FROM Kategorien A0
LEFT OUTER JOIN artikel A1 ON A0.Kategorie_Nr=A1.Kategorie_Nr
```

```
LEFT OUTER JOIN books A1E2 ON A0.Kategorie_Nr=A1E2.Kategorie_Nr
LEFT OUTER JOIN cds A1E1 ON A0.Kategorie_Nr=A1E1.Kategorie_Nr
GROUP BY A0.KategorieName
```

This SQL will select the columns "Lagerbestand" and "Einzelpreis" from one extent only, and for ProductGroups having Articles, Books and CDs the result is wrong!

As a workaround the query can be "reversed". Instead of selection the ProductGroup we go for the Articles:

```
Criteria crit = new Criteria();
Collection results = new Vector();
ReportQueryByCriteria q = QueryFactory.newReportQuery(Article.class, crit);
q.setAttributes(new String[] { "productGroup.groupName", "sum(stock)", "sum(price)" });
q.addGroupBy("productGroup.groupName");
```

This ReportQuery executes the following three selects (one for each concrete extent) and produces better results.

```
SELECT  A1.KategorieName,sum(A0.Lagerbestand),sum(A0.Einzelpreis)
FROM artikel A0
INNER JOIN Kategorien A1 ON A0.Kategorie_Nr=A1.Kategorie_Nr
GROUP BY A1.KategorieName

SELECT  A1.KategorieName,sum(A0.Lagerbestand),sum(A0.Einzelpreis)
FROM cds A0
INNER JOIN Kategorien A1 ON A0.Kategorie_Nr=A1.Kategorie_Nr
GROUP BY A1.KategorieName

SELECT  A1.KategorieName,sum(A0.Lagerbestand),sum(A0.Einzelpreis)
FROM books A0
INNER JOIN Kategorien A1 ON A0.Kategorie_Nr=A1.Kategorie_Nr
GROUP BY A1.KategorieName
```

Of course there's also a drawback here: the same ProductGroup may be selected several times, so to get the correct sum, the results of the ProductGroup has to be added. In our sample the ProductGroup "Books" will be listed three times.

After listing so many drawbacks and problems, here's the SQL the produces the desired result. This is a manually created SQL, not generated by OJB. Unfortunately it's not fully supported by some DBMS because of "union" and sub-selects.

```
select KategorieName, sum(lagerbestand), sum(einzelpreis)
from
(
        SELECT  A1.KategorieName,A0.Lagerbestand,A0.Einzelpreis
        FROM artikel A0
        INNER JOIN Kategorien A1 ON A0.Kategorie_Nr=A1.Kategorie_Nr

        union

        SELECT  A1.KategorieName,A0.Lagerbestand,A0.Einzelpreis
        FROM books A0
        INNER JOIN Kategorien A1 ON A0.Kategorie_Nr=A1.Kategorie_Nr

        union

        SELECT  A1.KategorieName,A0.Lagerbestand,A0.Einzelpreis
        FROM cds A0
        INNER JOIN Kategorien A1 ON A0.Kategorie_Nr=A1.Kategorie_Nr
)
group by kategorieName
```

3. ODMG OQL

4. JDO queries