Is your query too complex for JPA and Hibernate?

What you can do with JPQL

1. Define the attributes you want to select

Most developers use JPQL to select entities. But that's not the only projection you can use. You can define a list of entity attributes which you want to select as scalar values.

```
List<Object[]> authorNames = em.createQuery(
        “SELECT a.firstName, a.lastName FROM Author a”)
        .getResultList();
```

You can also use constructor expressions to define a constructor call.

```
List<BookPublisherValue> bookPublisherValues =
        em.createQuery(“SELECT new
org.thoughts.on.java.model.BookPublisherValue(b.title,
b.publisher.name) FROM Book b”,BookPublisherValue.class)
        .getResultList();
```

2. Join related entities in the FROM clause

There are 2 different ways to join related entities in JPQL queries. You can either create an implicit join by using the path operator in your *SELECT*, *WHERE*, *GROUP BY*, *HAVING* or *ORDER* clause:

```
em.createQuery(
        “SELECT b.title, b.publisher.name FROM Book b”)
        .getResultList();
```

or you define an explicit join in the FROM clause:

```
em.createQuery(
        “SELECT b.title, p.name FROM Book b JOIN
b.publisher p”)
        .getResultList();
```

I always recommend to define an explicit join in the FROM clause and not to mix the 2 approaches. Some older Hibernate versions generated 2 joins for the same relationship if you used implicit and explicit joins in the same JPQL statement. So better be careful.

3. Join unrelated entities in the FROM clause

Joining of unrelated entities is a Hibernate specific feature that I’m missing in JPA. Since Hibernate 5.1, you can join unrelated entities in a JPQL query. The syntax is very similar to SQL and I explained it in more detail in a previous post.

```
em.createQuery(
        “SELECT p.firstName, p.lastName, n.phoneNumber
FROM Person p JOIN PhoneBookEntry n ON p.firstName =
n.firstName AND p.lastName = n.lastName“)
        .getResultList();
```

4. Use conditional expressions in the WHERE and HAVING clause

JPQL supports a standard set of conditional expressions in the *WHERE* and *HAVING* clauses. You can use them to limit the result set to all Authors with an id equal to the given bind parameter value.

```
Query q = em.createQuery(
        “SELECT a FROM Author a WHERE a.id = :id”);
```

5. Use subqueries in the WHERE and HAVING clause

For some reason, JPQL’s support for subqueries seems to be a lesser known feature. It’s not as powerful as in SQL because it’s limited to the *WHERE* and *HAVING* clause, but you can use it at least there.

```
Query q = em.createQuery(
        “SELECT a FROM Author a WHERE a.id IN (SELECT
s.authorId FROM SpecialAuthors s)”);
```

6. Group your query results with GROUP BY and apply additional conditional expressions with HAVING

GROUP BY and *HAVING* are standard clauses in SQL, and that’s the same for JPQL. You can use them to group similar records in your result set and to apply additional conditional expressions on these groups.

```
em.createQuery(
        “SELECT a, count(b) FROM Author a JOIN a.books b
GROUP BY a”)
        .getResultList();
```

7. Order the query results with ORDER BY

ORDER BY is another JPQL clause that you know from SQL. You can use it to order the result of a query, and you should, of course, use it instead of ordering the result set in your Java code.

```
em.createQuery(
        “SELECT a FROM Author a ORDER BY a.lastName”)
        .getResultList();
```

8. Limit the number of records in your result set

The implementation of this feature feels a little bit strange if you’re used to the SQL syntax. JPQL doesn’t know the *LIMIT* keyword. You have to define the maximum number of returned rows on the Query interface and not in the JPQL statement. That has the benefit that you can do that in the same way for JPQL and Criteria API queries.

```
em.createQuery(“SELECT a FROM Author a”)
        .setMaxResults(10)
        .getResultList();
```

9. Use standard functions

JPQL also supports a small set of standard functions that you can use in your queries. You can use them to perform simple operations in the database instead of your Java code.

```
em.createQuery(
        “SELECT a, count(b) FROM Author a JOIN a.books b
GROUP BY a”)
        .getResultList();
```

10. Use non-standard and database specific functions

SQL supports more functions than JPQL and in addition to that, most databases provide a huge set of proprietary functions. Hibernate's database-specific dialects offer proprietary support for some of these functions and since JPA 2.1 you can call all functions supported by your database with a call of the *function* function.

```
em.createQuery(
        “SELECT a FROM Author a WHERE a.id = function(‘calculate’, 1, 2)“, Author.class)
        .getSingleResult();
```

11. Call stored procedures

JPA 2.1 also introduced the *@NamedStoredProcedureQuery* and the dynamic *StoredProcedureQuery* to provide basic support for stored procedure calls.

What you can't do with JPQL

As you've seen, JPQL supports a set of features that allows you to create queries up to a certain complexity. In my experience, these queries are good enough for most use cases. But if you want to implement reporting queries or have to rely on database-specific features, you will miss a lot of advanced SQL features. Here are a few of them that I miss on a regular basis.

1. Use subqueries outside of WHERE and HAVING clauses

That's the only features I often miss in JPQL and something that's annoying me for quite a while. With JPQL, you can use subqueries only in the *WHERE* and *HAVING* clauses but not in the *SELECT* and *FROM* clause.

SQL, of course, allows you to use subqueries also in the *SELECT* and *FROM* clause. In my experience, this is nothing you need on a daily basis, but I think I use it a few times in all of my projects.

2. Perform set operations

UNION, *INTERSECT*, and *EXCEPT* allow you to perform standard set operations on the result sets of independent queries. Lukas Eder explains them in detail in his blog post: You Probably don't Use SQL INTERSECT or EXCEPT Often Enough.

3. Use database specific hints

Most databases support proprietary query hints that allow you to provide additional information about your query. For some queries, the right set of hints can have a huge performance impact. You can learn about hints in Markus Winand's post: About Optimizer Hints.

4. Write recursive queries

Recursive queries are another nice SQL feature that allows you to traverse a graph of related database records.

5. Use window functions

If you don’t know about window functions in SQL, you have to watch one of Lukas Eder’s SQL talks or read some of his posts on the jOOQ blog. As soon as you understand this nifty SQL feature, you can do amazing things, like running total calculations or analyzing data series, with a relatively simple SQL statement.