Chapter 14. HQL: The Hibernate Query Language

Hibernate is equipped with an extremely powerful query language that (quite intentionally) looks very much like SQL. But don't be fooled by the syntax; HQL is fully object-oriented, understanding notions like inheritance, polymorphism and association.

14.1. Case Sensitivity

Queries are case-insensitive, except for names of Java classes and properties. So *Select* is the same as *SeleCT* is the same as *SELECT* but *org.hibernate.eg.FOO* is not *org.hibernate.eg.Foo* and *foo.barSet* is not *foo.BARSET*.

This manual uses lowercase HQL keywords. Some users find queries with uppercase keywords more readable, but we find this convention ugly when embedded in Java code.

14.2. The from clause

The simplest possible Hibernate query is of the form:

```
from eg.Cat
```

which simply returns all instances of the class *eg.Cat*. We don't usually need to qualify the class name, since auto-import is the default. So we almost always just write:

```
from Cat
```

Most of the time, you will need to assign an alias, since you will want to refer to the *Cat* in other parts of the query.

```
from Cat as cat
```

This query assigns the alias *cat* to *Cat* instances, so we could use that alias later in the query. The *as* keyword is optional; we could also write:

```
from Cat cat
```

Multiple classes may appear, resulting in a cartesian product or "cross" join.

```
from Formula, Parameter
```

```
from Formula as form, Parameter as param
```

It is considered good practice to name query aliases using an initial lowercase, consistent with Java naming standards for local variables (*eg*. *domesticCat*).

14.3. Associations and joins

We may also assign aliases to associated entities, or even to elements of a collection of values, using a join.
from Cat as cat 
    inner join cat.mate as mate 
    left outer join cat.kittens as kitten 

from Cat as cat left join cat.mate.kittens as kittens 

from Formula form full join form-parameter param 

The supported join types are borrowed from ANSI SQL 

- inner join 
- left outer join 
- right outer join 
- full join (not usually useful) 

The inner join, left outer join and right outer join constructs may be abbreviated. 

from Cat as cat 
    join cat.mate as mate 
    left join cat.kittens as kitten 

You may supply extra join conditions using the HQL with keyword. 

from Cat as cat 
    left join cat.kittens as kitten 
    with kitten.bodyWeight > 10.0 

In addition, a "fetch" join allows associations or collections of values to be initialized along with their parent objects, using a single select. This is particularly useful in the case of a collection. It effectively overrides the outer join and lazy declarations of the mapping file for associations and collections. See Section 19.1, “Fetching strategies” for more information. 

from Cat as cat 
    inner join fetch cat.mate 
    left join fetch cat.kittens

A fetch join does not usually need to assign an alias, because the associated objects should not be used in the where clause (or any other clause). Also, the associated objects are not returned directly in the query results. Instead, they may be accessed via the parent object. The only reason we might need an alias is if we are recursively join fetching a further collection: 

from Cat as cat 
    inner join fetch cat.mate 
    left join fetch cat.kittens child  
    left join fetch child.kittens 

Note that the fetch construct may not be used in queries called using iterate() (though scroll() can be used). Nor should fetch be used together with setMaxResults() or setFirstResult() as these operations are based on the result rows, which usually contain duplicates for eager collection fetching, hence, the number of rows is not what you'd expect. Nor may fetch be used together with an ad hoc with condition. It is possible to create a cartesian product by join fetching more than one collection in a query, so take care in this case. Join fetching multiple collection roles also sometimes gives unexpected results for bag mappings, so be careful about how you formulate your queries in this case. Finally, note that full join fetch and right join fetch are not meaningful.
If you are using property-level lazy fetching (with bytecode instrumentation), it is possible to force Hibernate to fetch the lazy properties immediately (in the first query) using _fetch all properties_.

```sql
from Document fetch all properties order by name
```

```sql
from Document doc fetch all properties where lower(doc.name) like '%cats%'
```

### 14.4. Forms of join syntax

HQL supports two forms of association joining: _implicit_ and _explicit_.

The queries shown in the previous section all use the _explicit_ form where the join keyword is explicitly used in the from clause. This is the recommended form.

The _implicit_ form does not use the join keyword. Instead, the associations are "dereferenced" using dot-notation. _implicit_ joins can appear in any of the HQL clauses. _implicit_ join result in inner joins in the resulting SQL statement.

```sql
from Cat as cat where cat.mate.name like '%s%'
```

### 14.5. Referring to identifier property

There are, generally speaking, 2 ways to refer to an entity's identifier property:

- The special property (lowercase) `id` may be used to reference the identifier property of an entity _provided that entity does not define a non-identifier property named id_.
- If the entity defines a named identifier property, you may use that property name.

References to composite identifier properties follow the same naming rules. If the entity has a non-identifier property named id, the composite identifier property can only be referenced by its defined named; otherwise, the special `id` property can be used to reference the identifier property.

Note: this has changed significantly starting in version 3.2.2. In previous versions, `id` _always_ referred to the identifier property no matter what its actual name. A ramification of that decision was that non-identifier properties named `id` could never be referenced in Hibernate queries.

### 14.6. The select clause

The _select_ clause picks which objects and properties to return in the query result set. Consider:

```sql
select mate
from Cat as cat
    inner join cat.mate as mate
```

The query will select mates of other Cats. Actually, you may express this query more compactly as:

```sql
select cat.mate from Cat cat
```

Queries may return properties of any value type including properties of component type:

```sql
select cat.name from DomesticCat cat
where cat.name like 'fri%'
```

```sql
select cust.name.firstName from Customer as cust
```
Queries may return multiple objects and/or properties as an array of type `Object[]`,

```java
select mother, offspr, mate.name
from DomesticCat as mother
    inner join mother.mate as mate
    left outer join mother.kittens as offspr
```

or as a `List`,

```java
select new list(mother, offspr, mate.name)
from DomesticCat as mother
    inner join mother.mate as mate
    left outer join mother.kittens as offspr
```

or as an actual typesafe Java object,

```java
select new Family(mother, mate, offspr)
from DomesticCat as mother
    join mother.mate as mate
    left join mother.kittens as offspr
```

assuming that the class `Family` has an appropriate constructor.

You may assign aliases to selected expressions using `as`:

```java
select max(bodyWeight) as max, min(bodyWeight) as min, count(*) as n
from Cat cat
```

This is most useful when used together with `select new map`:

```java
select new map( max(bodyWeight) as max, min(bodyWeight) as min, count(*) as n )
from Cat cat
```

This query returns a `Map` from aliases to selected values.

### 14.7. Aggregate functions

HQL queries may even return the results of aggregate functions on properties:

```java
select avg(cat.weight), sum(cat.weight), max(cat.weight), count(cat)
from Cat cat
```

The supported aggregate functions are

- `avg(...), sum(...), min(...), max(...)
- `count(*)`
- `count(...), count(distinct ...), count(all...)`

You may use arithmetic operators, concatenation, and recognized SQL functions in the select clause:

```java
select cat.weight + sum(kitten.weight)
from Cat cat
    join cat.kittens kitten
group by cat.id, cat.weight
```

```java
select firstName||' '||initial||' '||upper(lastName) from Person
```

The `distinct` and `all` keywords may be used and have the same semantics as in SQL.
14.8. Polymorphic queries

A query like:

```sql
from Cat as cat
```

returns instances not only of `Cat`, but also of subclasses like `DomesticCat`. Hibernate queries may name any Java class or interface in the `from` clause. The query will return instances of all persistent classes that extend that class or implement the interface. The following query would return all persistent objects:

```sql
from java.lang.Object o
```

The interface `Named` might be implemented by various persistent classes:

```sql
from Named n, Named m where n.name = m.name
```

Note that these last two queries will require more than one SQL `SELECT`. This means that the `order by` clause does not correctly order the whole result set. (It also means you can't call these queries using `Query.scroll()`.)

14.9. The `where` clause

The `where` clause allows you to narrow the list of instances returned. If no alias exists, you may refer to properties by name:

```sql
from Cat where name='Fritz'
```

If there is an alias, use a qualified property name:

```sql
from Cat as cat where cat.name='Fritz'
```

returns instances of `Cat` named 'Fritz'.

```sql
select foo
from Foo foo, Bar bar
where foo.startDate = bar.date
```

will return all instances of `Foo` for which there exists an instance of `bar` with a `date` property equal to the `startDate` property of the `Foo`. Compound path expressions make the `where` clause extremely powerful. Consider:

```sql
from Cat cat where cat.mate.name is not null
```

This query translates to an SQL query with a table (inner) join. If you were to write something like

```sql
from Foo foo
where foo.bar.baz.customer.address.city is not null
```

you would end up with a query that would require four table joins in SQL.

The `=` operator may be used to compare not only properties, but also instances:

```sql
from Cat cat, Cat rival where cat.mate = rival.mate
```
select cat, mate  
from Cat cat, Cat mate  
where cat.mate = mate  

The special property (lowercase) \texttt{id} may be used to reference the unique identifier of an object. See Section 14.5, “Referring to identifier property” for more information.  
from Cat as cat where cat.id = 123  
from Cat as cat where cat.mate.id = 69  

The second query is efficient. No table join is required!  

Properties of composite identifiers may also be used. Suppose \texttt{Person} has a composite identifier consisting of \texttt{country} and \texttt{medicareNumber}. Again, see Section 14.5, “Referring to identifier property” for more information regarding referencing identifier properties.  
from bank.Person person  
where person.id.country = 'AU'  
and person.id.medicareNumber = 123456  
from bank.Account account  
where account.owner.id.country = 'AU'  
and account.owner.id.medicareNumber = 123456  

Once again, the second query requires no table join.  

Likewise, the special property \texttt{class} accesses the discriminator value of an instance in the case of polymorphic persistence. A Java class name embedded in the where clause will be translated to its discriminator value.  
from Cat cat where cat.class = DomesticCat  

You may also use components or composite user types, or properties of said component types. See ??? for more details.  
An "any" type has the special properties \texttt{id} and \texttt{class}, allowing us to express a join in the following way (where \texttt{AuditLog.item} is a property mapped with \texttt{<any>}).  
from AuditLog log, Payment payment  
where log.item.class = 'Payment' and log.item.id = payment.id  

Notice that \texttt{log.item.class} and \texttt{payment.class} would refer to the values of completely different database columns in the above query.  

14.10. Expressions  

Expressions allowed in the where clause include most of the kind of things you could write in SQL:  

- mathematical operators +, -, *, /  
- binary comparison operators =, >=, <=, <>, !=, like  
- logical operations and, or, not  
- Parentheses ( ), indicating grouping  
- in, not in, between, is null, is not null, is empty, is not empty, member of and not member of
"Simple" case, case ... when ... then ... else ... end, and "searched" case, case when ... then ... else ... end

string concatenation ...||... or concat(...,...)

current_date(), current_time(), current_timestamp()

second(...), minute(...), hour(...), day(...), month(...), year(...),

Any function or operator defined by EJB-QL 3.0: substring(), trim(), lower(), upper(), length(), locate(), abs(), sqrt(), bit_length(), mod()

coaalesce() and nullif()

str() for converting numeric or temporal values to a readable string

cast(... as ...), where the second argument is the name of a Hibernate type, and extract(... from ...) if ANSI cast() and extract() is supported by the underlying database

the HQL index() function, that applies to aliases of a joined indexed collection

HQL functions that take collection-valued path expressions: size(), minelement(), maxelement(), minindex(), maxindex(), along with the special elements() and indices functions which may be quantified using some, all, exists, any, in.

Any database-supported SQL scalar function like sign(), trunc(), rtrim(), sin()

JDBC-style positional parameters ?

named parameters :name,:start_date,:x1

SQL literals 'foo', 69, 6.66E+2,'1970-01-01 10:00:01.0'

Java public static final constants eg.Color.TABBY

in and between may be used as follows:

from DomesticCat cat where cat.name between 'A' and 'B'

from DomesticCat cat where cat.name in ( 'Foo', 'Bar', 'Baz' )

and the negated forms may be written

from DomesticCat cat where cat.name not between 'A' and 'B'

from DomesticCat cat where cat.name not in ( 'Foo', 'Bar', 'Baz' )

Likewise, is null and is not null may be used to test for null values.

Booleans may be easily used in expressions by declaring HQL query substitutions in Hibernate configuration:

<property name="hibernate.query.substitutions">true 1, false 0</property>

This will replace the keywords true and false with the literals 1 and 0 in the translated SQL from this HQL:
from Cat cat where cat.alive = true

You may test the size of a collection with the special property size, or the special size() function.

from Cat cat where cat.kittens.size > 0

from Cat cat where size(cat.kittens) > 0

For indexed collections, you may refer to the minimum and maximum indices using minindex and maxindex functions. Similarly, you may refer to the minimum and maximum elements of a collection of basic type using the minelement and maxelement functions.

from Calendar cal where maxelement(cal.holidays) > current_date

from Order order where maxindex(order.items) > 100

from Order order where minelement(order.items) > 10000

The SQL functions any, some, all, exists, in are supported when passed the element or index set of a collection (elements and indices functions) or the result of a subquery (see below).

select mother from Cat as mother, Cat as kit
where kit in elements(foo.kittens)

select p from NameList list, Person p
where p.name = some elements(list.names)

from Cat cat where exists elements(cat.kittens)

from Player p where 3 > all elements(p.scores)

from Show show where 'fizard' in indices(show.acts)

Note that these constructs - size, elements, indices, minindex, maxindex, minelement, maxelement - may only be used in the where clause in Hibernate3.

Elements of indexed collections (arrays, lists, maps) may be referred to by index (in a where clause only):

from Order order where order.items[0].id = 1234

select person from Person person, Calendar calendar
where calendar.holidays['national day'] = person.birthDay
    and person.nationality.calendar = calendar

select item from Item item, Order order
where order.items[ order.deliveredItemIndices[0] ] = item and order.id = 11

select item from Item item, Order order
where order.items[ maxindex(order.items) ] = item and order.id = 11

The expression inside [] may even be an arithmetic expression.

select item from Item item, Order order
where order.items[ size(order.items) - 1 ] = item

HQL also provides the built-in index() function, for elements of a one-to-many association or
collection of values.

```sql
select item, index(item) from Order order
    join order.items item
where index(item) < 5
```

Scalar SQL functions supported by the underlying database may be used

```sql
from DomesticCat cat where upper(cat.name) like 'FRI%'
```

If you are not yet convinced by all this, think how much longer and less readable the following query would be in SQL:

```sql
select cust
from Product prod,
    Store store
    inner join store.customers cust
where prod.name = 'widget'
    and store.location.name in ( 'Melbourne', 'Sydney' )
    and prod = all elements(cust.currentOrder.lineItems)
```

**Hint:** something like

```sql
SELECT cust.name, cust.address, cust.phone, cust.id, cust.current_order
FROM customers cust,
    stores store,
    locations loc,
    store_customers sc,
    product prod
WHERE prod.name = 'widget'
    AND store.loc_id = loc.id
    AND loc.name IN ( 'Melbourne', 'Sydney' )
    AND sc.store_id = store.id
    AND sc.cust_id = cust.id
    AND prod.id = ALL(
        SELECT item.prod_id
            FROM line_items item, orders o
            WHERE item.order_id = o.id
        AND cust.current_order = o.id
    )
```

### 14.11. The order by clause

The list returned by a query may be ordered by any property of a returned class or components:

```sql
from DomesticCat cat
order by cat.name asc, cat.weight desc, cat.birthdate
```

The optional `asc` or `desc` indicate ascending or descending order respectively.

### 14.12. The group by clause

A query that returns aggregate values may be grouped by any property of a returned class or components:

```sql
select cat.color, sum(cat.weight), count(cat)
from Cat cat
group by cat.color
```

```sql
select foo.id, avg(name), max(name)
from Foo foo join foo.names name
```
A having clause is also allowed.

```sql
select cat.color, sum(cat.weight), count(cat)
from Cat cat
group by cat.color
having cat.color in (eg.Color.TABBY, eg.Color.BLACK)
```

SQL functions and aggregate functions are allowed in the having and order by clauses, if supported by the underlying database (eg. not in MySQL).

```sql
select cat
from Cat cat
  join cat.kittens kitten
group by cat.id, cat.name, cat.other, cat.properties
having avg(kitten.weight) > 100
order by count(kitten) asc, sum(kitten.weight) desc
```

Note that neither the group by clause nor the order by clause may contain arithmetic expressions. Also note that Hibernate currently does not expand a grouped entity, so you can't write group by cat if all properties of cat are non-aggregated. You have to list all non-aggregated properties explicitly.

### 14.13. Subqueries

For databases that support subselects, Hibernate supports subqueries within queries. A subquery must be surrounded by parentheses (often by an SQL aggregate function call). Even correlated subqueries (subqueries that refer to an alias in the outer query) are allowed.

```sql
from Cat as fatcat
where fatcat.weight > ( 
  select avg(cat.weight) from DomesticCat cat
)
```

```sql
from DomesticCat as cat
where cat.name = some ( 
  select name.nickName from Name as name
)
```

```sql
from Cat as cat
where not exists ( 
  from Cat as mate where mate.mate = cat
)
```

```sql
from DomesticCat as cat
where cat.name not in ( 
  select name.nickName from Name as name
)
```

```sql
select cat.id, (select max(kit.weight) from cat.kitten kit)
from Cat as cat
```

Note that HQL subqueries may occur only in the select or where clauses.

Note that subqueries can also utilize row value constructor syntax. See [Section 14.18, “Row value constructor syntax”](#) for more details.
14.14. HQL examples

Hibernate queries can be quite powerful and complex. In fact, the power of the query language is one of Hibernate's main selling points. Here are some example queries very similar to queries that I used on a recent project. Note that most queries you will write are much simpler than these!

The following query returns the order id, number of items and total value of the order for all unpaid orders for a particular customer and given minimum total value, ordering the results by total value. In determining the prices, it uses the current catalog. The resulting SQL query, against the ORDER, ORDER_LINE, PRODUCT, CATALOG and PRICE tables has four inner joins and an (uncorrelated) subselect.

```sql
select order.id, sum(price.amount), count(item)
from Order as order
    join order.lineItems as item
    join item.product as product,
    Catalog as catalog
    join catalog.prices as price
where order.paid = false
    and order.customer = :customer
    and price.product = product
    and catalog.effectiveDate < sysdate
    and catalog.effectiveDate >= all (
        select cat.effectiveDate
        from Catalog as cat
        where cat.effectiveDate < sysdate
    )
group by order
having sum(price.amount) > :minAmount
order by sum(price.amount) desc
```

What a monster! Actually, in real life, I'm not very keen on subqueries, so my query was really more like this:

```sql
select order.id, sum(price.amount), count(item)
from Order as order
    join order.lineItems as item
    join item.product as product,
    Catalog as catalog
    join catalog.prices as price
where order.paid = false
    and order.customer = :customer
    and price.product = product
    and catalog = :currentCatalog
group by order
having sum(price.amount) > :minAmount
order by sum(price.amount) desc
```

The next query counts the number of payments in each status, excluding all payments in the AWAITING_APPROVAL status where the most recent status change was made by the current user. It translates to an SQL query with two inner joins and a correlated subselect against the PAYMENT, PAYMENT_STATUS and PAYMENT_STATUS_CHANGE tables.

```sql
select count(payment), status.name
from Payment as payment
    join payment.currentStatus as status
    join payment.statusChanges as statusChange
where payment.status.name <> PaymentStatus.AWAITING_APPROVAL
    and 
    or (statusChange.timeStamp = (
        select max(change.timeStamp)
```
from PaymentStatusChange change
    where change.payment = payment
) and statusChange.user <> :currentUser
) group by status.name, status.sortOrder
order by status.sortOrder

If I would have mapped the statusChanges collection as a list, instead of a set, the query would have been much simpler to write.

select count(payment), status.name
from Payment as payment
    join payment.currentStatus as status
where payment.status.name <> PaymentStatus.AWAITING_APPROVAL
    or payment.statusChanges[ maxIndex(payment.statusChanges) ].user <> :currentUser
) group by status.name, status.sortOrder
order by status.sortOrder

The next query uses the MS SQL Server isNull() function to return all the accounts and unpaid payments for the organization to which the current user belongs. It translates to an SQL query with three inner joins, an outer join and a subselect against the ACCOUNT, PAYMENT, PAYMENT_STATUS, ACCOUNT_TYPE, ORGANIZATION and ORG_USER tables.

select account, payment
from Account as account
    left outer join account.payments as payment
where :currentUser in elements(account.holder.users)
    and PaymentStatus.UNPAID = isNull(payment.currentStatus.name, PaymentStatus.UNPAID)
order by account.type.sortOrder, account.accountNumber, payment.dueDate

For some databases, we would need to do away with the (correlated) subselect.

select account, payment
from Account as account
    join account.holder.users as user
    left outer join account.payments as payment
where :currentUser = user
    and PaymentStatus.UNPAID = isNull(payment.currentStatus.name, PaymentStatus.UNPAID)
order by account.type.sortOrder, account.accountNumber, payment.dueDate

14.15. Bulk update and delete

HQL now supports update, delete and insert ... select ... statements. See Section 13.4, “DML-style operations” for details.

14.16. Tips & Tricks

You can count the number of query results without actually returning them:

( (Integer) session.iterate("select count(*) from ....").next() ).intValue()

To order a result by the size of a collection, use the following query:

select usr.id, usr.name
from User as usr
    left join usr.messages as msg
group by usr.id, usr.name
order by count(msg)

If your database supports subselects, you can place a condition upon selection size in the where clause of your query:
from User usr where size(usr.messages) >= 1

If your database doesn't support subselects, use the following query:
select usr.id, usr.name
from User usr
    join usr.messages msg
group by usr.id, usr.name
having count(msg) >= 1

As this solution can't return a User with zero messages because of the inner join, the following form is also useful:
select usr.id, usr.name
from User as usr
    left join usr.messages as msg
group by usr.id, usr.name
having count(msg) = 0

Properties of a JavaBean can be bound to named query parameters:
Query q = s.createQuery("from foo Foo as foo where foo.name=:name and foo.size=:size");
q.setProperties(fooBean); // fooBean has getName() and getSize()
List foos = q.list();

Collections are pageable by using the Query interface with a filter:
Query q = s.createFilter( collection, "" ); // the trivial filter
q.setMaxResults(PAGE_SIZE);
q.setFirstResult(PAGE_SIZE * pageNumber);
List page = q.list();

Collection elements may be ordered or grouped using a query filter:
Collection orderedCollection = s.filter( collection, "order by this.amount" );
Collection counts = s.filter( collection, "select this.type, count(this) group by this.type" );

You can find the size of a collection without initializing it:
( (Integer) session.iterate("select count(*) from ....").next() ).intValue();

14.17. Components
Components might be used in just about every way that simple value types can be used in HQL queries. They can appear in the select clause:
select p.name from from Person p
select p.name.first from from Person p

where the Person's name property is a component. Components can also be used in the where clause:
Components can also be used in the order by clause:

from from Person p order by p.name

from from Person p order by p.name.first

Another common use of components is in **Section 14.18, “Row value constructor syntax”**.

### 14.18. Row value constructor syntax

HQL supports the use of ANSI SQL *row value constructor* syntax (sometimes called *tuple* syntax), even though the underlying database may not support that notion. Here we are generally referring to multi-valued comparisons, typically associated with components. Consider an entity `Person` which defines a name component:

```sql
from Person p where p.name.first='John' and p.name.last='Jingleheimer-Schmidt'
```

That's valid syntax, although a little verbose. It be nice to make this a bit more concise and use *row value constructor* syntax:

```sql
from Person p where p.name=('John', 'Jingleheimer-Schmidt')
```

It can also be useful to specify this in the select clause:

```sql
select p.name from from Person p
```

Another time using *row value constructor* syntax can be beneficial is when using subqueries needing to compare against multiple values:

```sql
from Cat as cat
where not ( cat.name, cat.color ) in (  
    select cat.name, cat.color from DomesticCat cat
)
```

One thing to consider when deciding if you want to use this syntax is that the query will be dependent upon the ordering of the component sub-properties in the metadata.