# Chapter 14. HQL: The Hibernate Query Language

Hibernate uses a powerful query language (HQL) that is similar in appearance to SQL. Compared with SQL, however, HQL is fully object-oriented and understands notions like inheritance, polymorphism and association.

## 14.1. Case Sensitivity

With the exception of names of Java classes and properties, queries are case-insensitive. 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 this convention is unsuitable for queries embedded in Java code.

## 14.2. The from clause

The simplest possible Hibernate query is of the form:

```
from eg.Cat
```

This returns all instances of the class eg.Cat. You do not usually need to qualify the class name, since auto-import is the default. For example:

```
from Cat
```

In order to refer to the Cat in other parts of the query, you will need to assign an *alias*. For example:

```
from Cat as cat
```

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

```
from Cat cat
```

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

```
from Formula, Parameter
```

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

It is good practice to name query aliases using an initial lowercase as this is consistent with Java naming standards for local variables (e.g. domesticCat).

## 14.3. Associations and joins

You can also assign aliases to associated entities or to elements of a collection of values using a join. For example:

```
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
```

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). The associated objects are also not returned directly in the query results. Instead, they may be accessed via the parent object. The only reason you might need an alias is if you 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
```

The fetch construct cannot be used in queries called using iterate() (though scroll() can be used). Fetch should 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 would expect. Fetch should also not be used together with impromptu 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 can produce unexpected results for bag mappings, so user discretion is advised when formulating 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 in the first query immediately using fetch all properties.

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

```
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, that is, 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.

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

## 14.5. Referring to identifier property

There are 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 the entity does not define a non-identifier property named id*.
- If the entity defines a named identifier property, you can 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.

**Important**

Please note that, starting in version 3.2.2, this has changed significantly. In previous versions, id *always* referred to the identifier property regardless of 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 the following:

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

The query will select mates of other Cats. You can express this query more compactly as:

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

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

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

```
select cust.name.firstName from Customer as cust
```

Queries can return multiple objects and/or properties as an array of type Object[]:

```
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:

```
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 - assuming that the class Family has an appropriate constructor - as an actual typesafe Java object:

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

You can assign aliases to selected expressions using as:

```
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:

```
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 can even return the results of aggregate functions on properties:

```
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 can use arithmetic operators, concatenation, and recognized SQL functions in the select clause:

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

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

The distinct and all keywords can be used and have the same semantics as in SQL.

```
select distinct cat.name from Cat cat

select count(distinct cat.name), count(cat) from Cat cat
```

## 14.8. Polymorphic queries

A query like:

```
from Cat as cat
```

returns instances not only of Cat, but also of subclasses like DomesticCat. Hibernate queries can 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:

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

The interface Named might be implemented by various persistent classes:

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

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 cannot call these queries using Query.scroll().

## 14.9. The where clause

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

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

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

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

This returns instances of Cat named 'Fritz'.

The following query:

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

returns all instances of Foo with 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 the following:

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

This query translates to an SQL query with a table (inner) join. For example:

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

would result in a query that would require four table joins in SQL.

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

```
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) id can 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 and does not require a table join.

Properties of composite identifiers can also be used. Consider the following example where Person has composite identifiers consisting of country and medicareNumber:

```
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 does not require a table join.

See Section 14.5, “Referring to identifier property” for more information regarding referencing identifier properties)

The special property 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 can also use components or composite user types, or properties of said component types. See

Section 14.17, “Components” for more information.

An "any" type has the special properties id and class that allows you to express a join in the following way (where AuditLog.item is a property mapped with <any>):

```
from AuditLog log, Payment payment
where log.item.class = 'Payment' and log.item.id = payment.id
```

The log.item.class and payment.class would refer to the values of completely different database columns in the above query.

## 14.10. Expressions

Expressions used in the where clause include the following:

- mathematical operators: +, -, *, /
- binary comparison operators: =, >=, <=, <>, !=, like
- logical operations and, or, not
- Parentheses ( ) that indicates 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(), and current_timestamp()
- second(...), minute(...), hour(...), day(...), month(...), and year(...)
- Any function or operator defined by EJB-QL 3.0: substring(), trim(), lower(), upper(), length(), locate(), abs(), sqrt(), bit_length(), mod()
- coalesce() 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 that can be quantified using some, all, exists, any, in.
- Any database-supported SQL scalar function like sign(), trunc(), rtrim(), and sin()
- JDBC-style positional parameters ?
- named parameters :name, :start_date, and :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 can be used as follows:

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

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

The negated forms can be written as follows:

```
from DomesticCat cat where cat.name not between 'A' and 'B'
from DomesticCat cat where cat.name not in ( 'Foo', 'Bar', 'Baz' )
```

Similarly, is null and is not null can be used to test for null values.

Booleans can 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 can 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 can refer to the minimum and maximum indices using minindex and maxindex functions. Similarly, you can refer to the minimum and maximum elements of a collection of basic type using the minelement and maxelement functions. For example:

```
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 – can only be used in the where clause in Hibernate3.

Elements of indexed collections (arrays, lists, and maps) can 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 [] can 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.

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

Scalar SQL functions supported by the underlying database can be used:

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

Consider how much longer and less readable the following query would be in 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

```
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 can be ordered by any property of a returned class or components:

```
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 can be grouped by any property of a returned class or components:

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

```
select foo.id, avg(name), max(name)
from Foo foo join foo.names name
group by foo.id
```

A having clause is also allowed.

```
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 they are supported by the underlying database (i.e., not in MySQL).

```
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
```

Neither the group by clause nor the order by clause can contain arithmetic expressions. Hibernate also does not currently expand a grouped entity, so you cannot 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.

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

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

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

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

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

Note that HQL subqueries can 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 information.

## 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 strengths. The following example queries are similar to queries that have been used on recent projects. Please note that most queries you will write will be much simpler than the following examples.

The following query returns the order id, number of items, the given minimum total value and the total value of the order for all unpaid orders for a particular customer. The results are ordered 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.

```
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:

```
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.

```
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
    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 the statusChanges collection was mapped 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 more information.

## 14.16. Tips & Tricks

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

```
( (Integer) session.createQuery("select count(*) from ....").iterate().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 does not support subselects, use the following query:

```
select usr.id, usr.name
from User usr.name
    join usr.messages msg
group by usr.id, usr.name
having count(msg) >= 1
```

As this solution cannot 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 can 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.createQuery("select count(*) from ....").iterate().next() ).intValue();
```

## 14.17. Components

Components can be used similarly to the simple value types that are used in HQL queries. They can appear in the select clause as follows:

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

```
select p.name.first from Person p
```

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

```
from Person p where p.name = :name
```

```
from Person p where p.name.first = :firstName
```

Components can also be used in the order by clause:

```
from Person p order by p.name
```

```
from Person p order by p.name.first
```

Another common use of components is in row value constructors.

## 14.18. Row value constructor syntax

HQL supports the use of ANSI SQL row value constructor syntax, sometimes referred to AS 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:

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

That is valid syntax although it is a little verbose. You can make this more concise by using row value constructor syntax:

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

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

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

Using row value constructor syntax can also be beneficial when using subqueries that need to compare against multiple values:

```
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.