A Guide to Hibernate Query Language

Version 7.0.0-SNAPSHOT

Table of Contents

Preface

Hibernate 6 was a major redesign of the world's most popular and feature-rich ORM solution. The redesign touched almost every subsystem of Hibernate, including the APIs, mapping annotations, and, above all else, the query language. This was the second time Hibernate Query Language had been completely reimplemented from scratch, but the first time in more than fifteen years. Hibernate 7 rests on this foundation.

In this new incarnation, HQL is far more powerful, and the HQL compiler much more robust. Today, HQL has a feature set to match that of modern dialects of SQL, and is able to take full advantage of the power of modern SQL databases.

This document is a reference guide to the full feature set of the language, and is the only up-to-date source for those who wish to learn how to write HQL effectively in Hibernate 6 and Hibernate 7.

If you are unfamiliar with Hibernate, be sure to first read the Short Guide to Hibernate.

Chapter 1. Basic concepts

This document describes Hibernate Query Language (HQL), which is, I suppose we could say, a "dialect" of the Java (now Jakarta) Persistence Query Language (JPQL).

Or is it the other way around?

JPQL was inspired by early versions of HQL, and is a proper subset of modern HQL. Here we focus on describing the complete, more powerful HQL language as it exists today.

If strict JPA compliance is what you're looking for, use the setting `hibernate.jpa.compliance.query=true`. With this configuration, any attempt to use HQL features beyond the JPQL subset will result in an exception.

We don't recommend the use of this setting.

The truth is that HQL today has capabilities that go far beyond what is possible in plain JPQL. We're not going to fuss too much about not limiting ourselves to the standard here. Faced with a choice between writing database-specific native SQL, or database-independent HQL, we know what our preference is.

1.1. HQL and SQL

Throughout this document, we'll assume you know SQL and the relational model, at least at a basic level. HQL and JPQL are loosely based on SQL and are easy to learn for anyone familiar with SQL.

For example, if you understand this SQL query:

```
select book.title, pub.name            /* projection */
from Book as book                      /* root table */
    join Publisher as pub              /* table join */
        on book.publisherId = pub.id   /* join condition */
where book.title like 'Hibernate%'     /* restriction (selection) */
order by book.title                    /* sorting */
```

Then we bet you can already make sense of this HQL:

```
select book.title, pub.name            /* projection */
from Book as book                      /* root entity */
    join book.publisher as pub         /* association join */
where book.title like 'Hibernate%'     /* restriction (selection) */
order by book.title                    /* sorting */
```

You might notice that even for this very simple example, the HQL version is slightly shorter. This is typical. Actually, HQL queries are usually much more compact than the SQL they compile to.

But there's one huge difference: in HQL, `Book` refers to an entity class written in Java, and `book.title` to a field of that class. We're not permitted to directly reference database tables and columns in HQL or JPQL.

In this chapter, we'll demonstrate how similar HQL is to SQL by giving a quick overview of the basic statement types. You'll be bored to discover they're exactly the ones you expect: `select`, `insert`, `update`, and `delete`.

This is a reference guide. We're not going to explain basic concepts like ternary logic, joins, aggregation, selection, or projection, because that information is freely available elsewhere, and anyway we couldn't possibly do these topics justice here. If you don't have a firm grasp of these ideas, it's time to pick up a book about SQL or about the relational model.

But first we need to mention something that's a bit different to SQL. HQL has a slightly complicated way of dealing with case sensitively.

1.2. Lexical structure

Lexically, JPQL is quite similar to SQL, so in this section we'll limit ourselves to mentioning those places where it differs.

1.2.1. Identifiers and case sensitivity

An identifier is a name used to refer to an entity, an attribute of a Java class, an identification variable, or a function.

For example, `Book`, `title`, `author`, and `upper` are all identifiers, but they refer to different kinds of things. In HQL and JPQL, the case sensitivity of an identifier depends on the kind of thing the identifier refers to.

The rules for case sensitivity are:

- keywords and function names are case-insensitive, but
- identification variable names, Java class names, and the names of attributes of Java classes, are case-sensitive.

We apologize for this inconsistency. In hindsight, it might have been better to define the whole language as case-sensitive.

Incidentally, it's standard practice to use lowercase keywords in HQL.

The use of uppercase keywords indicates an endearing but unhealthy attachment to the culture of the 1970's.

Just to reiterate these rules:

`select`, `SeLeCT`, `sELEct`, and `SELECT`	All the same, `select` is a keyword
`upper(name)` and `UPPER(name)`	Same, `upper` is a function name
`from BackPack` and `from Backpack`	Different, refer to different Java classes
`person.nickName` and `person.nickname`	Different, since the path expression element `nickName` refers to an attribute of an entity defined in Java
`person.nickName`, `Person.nickName`, and `PERSON.nickName`	All different, since the first element of a path expression is an identification variable

The JPQL specification defines identification variables as case-*insensitive*. And so in strict JPA-compliant mode, Hibernate treats `person.nickName`, `Person.nickName`, and `PERSON.nickName` as the *same*.

A *quoted identifier* is written in backticks. Quoting lets you use a keyword as an identifier.

```
select thing.interval.`from` from Thing thing
```

Actually, in most contexts, HQL keywords are "soft", and don't need to be quoted. The parser is usually able to distinguish if the reserved word is being used as a keyword or as an identifier.

1.2.2. Comments

Comments in HQL look like multiline comments in Java. They're delimited by `/*` and `*/`.

Neither SQL-style `--` nor Java-style `//` line-ending comments are allowed.

It's quite rare to see comments in HQL, but perhaps it will be more common now that Java has text blocks.

1.2.3. Parameters

Parameters come in two flavors in JPQL, and HQL supports a third flavor for historical reasons:

Parameter type	Examples	Usage from Java
Named parameters	`:name`, `:title`, `:id`	`query.setParameter("name", name)`
Ordinal parameters	`?1`, `?2`, `?3`	`query.setParameter(1, name)`
JDBC-style parameters 💀	`?`	`query.setParameter(1, name)`

JDBC-style parameters of form `?` are like ordinal parameters where the index is inferred from the position in the text of the query. JDBC-style parameters are deprecated.

It's *extremely* important to use parameters to pass user input to the database. Constructing a query by concatenating HQL fragments with user input is extremely dangerous, opening the door to the possibility of executing arbitrary code on the database server.

1.2.4. Literals

Some of the syntax for literal values also departs from the standard syntax in ANSI SQL, especially in the area of date/time literals, but we'll discuss all that later, in Literals.

1.3. Syntax

We'll describe the syntax of the language as we go along, sometimes displaying fragments of the grammar in an ANTLR-like BNF form. (Occasionally we'll simplify these snippets for readability, so please don't take them as canonical.)

The full canonical grammar for HQL can be found in the github project.

The grammar for JPQL may be found in chapter 4 of the JPA specification.

1.4. Type system

JPA doesn't have a well-specified type system, but, reading between the lines a bit, the following types may be discerned:

- entity types,
- numeric values,
- strings,
- dates/times,
- booleans, and
- enumerated types.

Such a coarse-grained type system is in some sense an insufficient constraint on implementors of the specification, or, viewed from a different perspective, it leaves us quite a lot of flexibility.

The way HQL interprets this type system is to assign a Java type to every expression in the language. Thus, numeric expressions have types like `Long`, `Float`, or `BigInteger`, date/time expressions have types like `LocalDate`, `LocalDateTime`, or `Instant`, and boolean expressions are always of type `Boolean`.

Going further, an expression like `local datetime - document.created` is assigned the Java type `java.time.Duration`, a type which doesn't appear anywhere in the JPA specification.

Since the language must be executed on SQL databases, every type accommodates null values.

1.4.1. Null values and ternary logic

The SQL `null` behaves quite differently to a null value in Java.

- In Java, an expression like `number + 1` produces in an exception if `number` is null.
- But in SQL, and therefore also in HQL and JPQL, such an expression evaluates to `null`.

> It's almost always the case that an operation applied to a null value yields another null value. This rule applies to function application, to operators like `*` and `||`, to comparison operators like `<` and `=`, and even to logical operations like `and` and `not`.
>
> The exceptions to this rule are the `is null` operator and the functions `coalesce()` and `ifnull()` which are specifically designed for dealing with null values.

This rule is the source of the famous (and controversial) *ternary logic* of SQL. A logical expression like `firstName='Gavin' and team='Hibernate'` isn't restricted to the values `true` and `false`. It may also be `null`.

This can, in principle, lead to some quite unintuitive results: we can't use the law of the excluded middle to reason about logical expressions in SQL! But in practice, we've never once run into a case where this caused us problems.

As you probably know, when a logical predicate occurs as a restriction, rows for which the predicate evaluates to `null` are *excluded* from the result set. That is, in this context at least, a logical null is interpreted as "effectively false".

1.5. Statement types

HQL features four different kinds of statement:

- `select` queries,
- `update` statements,
- `delete` statements, and

- `insert … values` and `insert … select` statements.

Collectively, `insert`, `update`, and `delete` statements are sometimes called *mutation queries*. We need to be a little bit careful when executing mutation queries via a stateful session.

> The effect of an `update` or `delete` statement is not reflected in the persistence context, nor in the state of entity objects held in memory at the time the statement is executed.
>
> It's the responsibility of the client program to maintain synchronization of state held in memory with the database after execution of an `update` or `delete` statement.

Let's consider each type of mutation query in turn, beginning with the most useful type.

1.5.1. Update statements

The BNF for an `update` statement is quite straightforward:

```
updateStatement
    : "UPDATE" "VERSIONED"? targetEntity setClause whereClause?

targetEntity
    : entityName variable?

setClause
    : "SET" assignment ("," assignment)*

assignment
    : simplePath "=" expression
```

The `set` clause has a list of assignments to attributes of the given entity.

For example:

```
update Person set nickName = 'Nacho' where name = 'Ignacio'
```

Update statements are polymorphic, and affect mapped subclasses of the given entity class. Therefore, a single HQL `update` statement might result in multiple SQL update statements executed against the database.

An `update` statement must be executed using `Query.executeUpdate()`.

```
// JPA API
int updatedEntities = entityManager.createQuery(
        "update Person p set p.name = :newName where p.name = :oldName")
            .setParameter("oldName", oldName)
            .setParameter("newName", newName)
            .executeUpdate();


// Hibernate native API
int updatedEntities = session.createMutationQuery(
        "update Person set name = :newName where name = :oldName")
            .setParameter("oldName", oldName)
            .setParameter("newName", newName)
            .executeUpdate();
```

The integer value returned by `executeUpdate()` indicates the number of entity instances affected by the operation.

> In a `JOINED` inheritance hierarchy, multiple rows are required to store a single entity instance. In this case, the update count returned by Hibernate might not be exactly the same as the number of rows affected in the database.

An `update` statement, by default, does not affect the column mapped by the `@Version` attribute of the affected entities.

Adding the keyword `versioned`—writing `update versioned`—specifies that Hibernate should increment the version number or update the last modification timestamp.

```
update versioned Book set title = :newTitle where ssn = :ssn
```

An `update` statement may not directly `join` other entities, but it may:

- have an implicit join, or
- have subqueries in its `set` clause, or in its `where` clause, and the subqueries may contain joins.

1.5.2. Delete statements

The BNF for a `delete` statement is even simpler:

```
deleteStatement
    : "DELETE" "FROM"? targetEntity whereClause?
```

For example:

```
delete Author author where author.books is empty
```

As in SQL, the presence or absence of the `from` keyword has absolutely no effect on the semantics of the `delete` statement.

Just like update statements, delete statements are polymorphic, and affect mapped subclasses of the given entity class. Therefore, a single HQL `delete` statement might result in multiple SQL delete statements executed against the database.

A `delete` statement is executed by calling `Query.executeUpdate()`.

The integer value returned by `executeUpdate()` indicates the number of entity instances affected by the operation.

A `delete` statement may not directly `join` other entities, but it may:

- have an implicit join, or
- have subqueries in its `where` clause, and the subqueries may contain joins.

1.5.3. Insert statements

There are two kinds of `insert` statement:

- `insert … values`, where the attribute values to insert are given directly as tuples, and
- `insert … select`, where the inserted attribute values are sourced from a subquery.

The first form inserts a single row in the database, or multiple rows if you provide multiple tuples in the `values` clause. The second form may insert many new rows, or none at all.

The first sort of `insert` statement is not as useful. It's usually better to just use `persist()`.

But you might consider using it to set up test data.

`insert` statements are not part of JPQL.

The BNF for an `insert` statement is:

```
insertStatement
    : "INSERT" "INTO"? targetEntity targetFields
      (queryExpression | valuesList)
      conflictClause?

targetEntity
    : entityName variable?

targetFields
    : "(" simplePath ("," simplePath)* ")"

valuesList
    : "VALUES" values ("," values)*

values
    : "(" expression ("," expression)* ")"
```

For example:

```
insert Person (id, name)
    values (100L, 'Jane Doe'), (200L, 'John Roe')


insert into Author (id, name, bio)
    select id, name, name || ' is a newcomer for ' || str(year(local date))
    from Person
    where id = :pid
```

As in SQL, the presence or absence of the `into` keyword has no effect on the semantics of the `insert` statement.

From these examples we might notice that `insert` statements are in one respect a bit different to `update` and `delete` statements.

An `insert` statement is inherently *not* polymorphic! Its list of target fields is of fixed length, whereas each subclass of an entity class might declare additional fields. If the entity is involved in a mapped inheritance hierarchy, only attributes declared directly by the named entity and its superclasses may occur in the list of target fields. Attributes declared by subclasses may not occur.

The `queryExpression` in an `insert … select` statement may be any valid `select` query, with the caveat that the types of the values in the `select` list must match the types of the target fields.

This is checked during query compilation rather than allowing the type check to delegate to the database. This may cause problems when two Java types map to the same database type. For example, an attribute of type `LocalDateTime` and an attribute of type `Timestamp` both map to the SQL type `timestamp`, but are not considered assignable by the query compiler.

There are two ways to assign a value to the `@Id` attribute:

- explicitly specify the id attribute in the list of target fields, and its value in the values assigned to the target fields, or
- omit it, in which case a generated value is used.

Of course, the second option is only available for entities with database-level id generation (sequences or identity/autoincrement columns). It's not available for entities whose id generator is implemented in Java, nor for entities whose id is assigned by the application.

The same two options are available for a `@Version` attribute. When no version is explicitly specified, the version for a new entity instance is used.

The `on conflict` clause lets us specify what action should be taken when the database already contains the record we're attempting to insert.

```
conflictClause
    : ON CONFLICT conflictTarget? "DO" conflictAction

conflictTarget
    : ON CONSTRAINT identifier
    | "(" simplePath ("," simplePath)* ")"

conflictAction
    : "NOTHING"
    | "UPDATE" setClause whereClause?
```

Note that the `on constraint` variant accepting the name of a unique constraint only works on certain databases, or when just a single row is being inserted.

```
insert Person (ssn, name, phone)
    values ('116-76-1234', 'Jane Doe', '404 888 4319')
    on conflict (ssn) do update
        set phone = excluded.phone
```

Like `update` and `delete` statements, an `insert` statement must be executed by calling `Query.executeUpdate()`.

Now it's time to look at something *much* more complicated.

1.5.4. Select statements

Select statements retrieve and analyse data. This is what we're really here for.

The full BNF for a `select` query is quite complicated, but there's no need to understand it now. We're displaying it here for future reference.

```
selectStatement
    : queryExpression

queryExpression
    : withClause? orderedQuery (setOperator orderedQuery)*

orderedQuery
    : (query | "(" queryExpression ")") queryOrder?

query
    : selectClause fromClause? whereClause? (groupByClause havingClause?)?
    | fromClause whereClause? (groupByClause havingClause?)? selectClause?
    | whereClause

queryOrder
    : orderByClause limitClause? offsetClause? fetchClause?

fromClause
    : "FROM" entityWithJoins ("," entityWithJoins)*

entityWithJoins
    : fromRoot (join | crossJoin | jpaCollectionJoin)*

fromRoot
    : entityName variable?
    | "LATERAL"? "(" subquery ")" variable?

join
    : joinType "JOIN" "FETCH"? joinTarget joinRestriction?

joinTarget
    : path variable?
    | "LATERAL"? "(" subquery ")" variable?

withClause
    : "WITH" cte ("," cte)*
    ;
```

Most of the complexity here arises from the interplay of set operators (`union`, `intersect`, and `except`) with sorting.

We'll describe the various clauses of a query later, in Root entities and joins and in Selection, projection, and aggregation, but for now, to summarize, a query might have these bits:

Clause	Jargon	Purpose
`with`	Common table expressions	Declares named subqueries to be used in the following query
`from` and `join`	Roots and joins	Specifies the entities involved in the query, and how they're related to each other
`where`	Selection/restriction	Specifies a restriction on the data returned by the query
`group by`	Aggregation/grouping	Controls aggregation
`having`	Selection/restriction	Specifies a restriction to apply *after* aggregation
`select`	Projection	Specifies a projection (the things to return from the query)
`union`, `intersect`, `except`	Set algebra	These are set operators applied to the results of multiple subqueries
`order by`	Ordering	Specifies how the results should be sorted
`limit`, `offset`, `fetch`	Limits	Allows for limiting or paginating the results

Every one of these clauses is optional!

For example, the simplest query in HQL has no `select` clause at all:

```
from Book
```

But we don't necessarily *recommend* leaving off the `select` list.

HQL doesn't require a `select` clause, but JPQL *does*.

Naturally, the previous query may be written with a `select` clause:

```
select book from Book book
```

But when there's no explicit `select` clause, the select list is implied by the result type of the query:

```
// result type Book, only the Book selected
List<Book> books =
        session.createQuery("from Book join authors", Book.class)
            .getResultList();
for (Book book: books) {
    ...
}


// result type Object[], both Book and Author selected
List<Object[]> booksWithAuthors =
       session.createQuery("from Book join authors", Book.class, Object[].class)
            .getResultList();
for (var bookWithAuthor: booksWithAuthors) {
    Book book = (Book) bookWithAuthor[0];
    Author author = (Author) bookWithAuthor[1];
    ...
}
```

For complicated queries, it's probably best to explicitly specify a `select` list.

An alternative "simplest" query has *only* a `select` list:

```
select local datetime
```

This results in a SQL `from dual` query (or equivalent).

Looking carefully at the BNF given above, you might notice that the `select` list may occur either at the beginning of a query, or near the end, right before `order by`.

Of course, standard SQL, and JPQL, require that the `select` list comes at the beginning. But it's more natural to put it last:

```
from Book book select book.title, book.isbn
```

This form of the query is more readable, because the alias is declared *before* it's used, just as God and nature intended.

Naturally, queries are always polymorphic. Indeed, a fairly innocent-looking HQL query can easily translate to a SQL statement with many joins and unions.

We need to be a *bit* careful about that, but actually it's usually a good thing. HQL makes it very easy to fetch all the data we need in a single trip to the database, and that's absolutely key to achieving high performance in data access code. Typically, it's much worse to fetch exactly the data we need, but in many round trips to the database server, than it is to fetch just a bit more data than what we're going to need, all in a single SQL query.

When there's no explicit `select` clause, a further abbreviation is sometimes possible.

When the result type of a `select` query is an entity type, and we specify the type explicitly by passing the entity class to `createQuery()` or `createSelectionQuery()`, we're sometimes allowed to omit the `from` clause, for example:

```
// explicit result type Book, so 'from Book' is inferred
List<Book> books =
        session.createQuery("where title like :title", Book.class)
            .setParameter("title", title)
            .getResultList();
```

1.6. Representing result sets in Java

One of the most uncomfortable aspects of working with data in Java is that there's no good way to represent a table. Languages designed for working with data—R is an excellent example—always feature some sort of built-in table or "data frame" type. Of course, Java's type system gets in the way here. This problem is much easier to solve in a dynamically-typed language. The fundamental problem for Java is that it doesn't have tuple types.

Queries in Hibernate return tables. Sure, often a column holds whole entity objects, but we're not restricted to returning a single entity, and we often write queries that return multiple entities in each result, or which return things which aren't entities.

So we're faced with the problem of representing such result sets, and, we're sad to say, there's no fully general and completely satisfying solution.

Let's begin with the easy case.

1.6.1. Queries with a single projected item

If there's just one projected item in the `select` list, then, no sweat, that's the type of each query result.

```
List<String> results =
        entityManager.createQuery("select title from Book", String.class)
            .getResultList();
```

There's really no need to fuss about with trying to represent a "tuple of length 1". We're not even sure what to call those.

Problems arise as soon as we have multiple items in the `select` list of a query.

1.6.2. Queries with multiple projected items

When there are multiple expressions in the select list then, by default, and in compliance with JPA, each query result is packaged as an array of type `Object[]`.

```
List<Object[]> results =
        entityManager.createQuery("select title, left(text, 200) from Book",
                                  Object[].class)
            .getResultList();
for (var result : results) {
    String title = (String) result[0];
    String preamble = (String) result[1];
}
```

This is bearable, but let's explore some other options.

JPA lets us specify that we want each query result packaged as an instance of `jakarta.persistence.Tuple`. All we have to do is pass the class `Tuple` to `createQuery()`.

```
List<Tuple> tuples =
        entityManager.createQuery("select title as title, left(text, 200) as preamble from Book",
                                  Tuple.class)
            .getResultList();
for (Tuple tuple : tuples) {
    String title = tuple.get("title", String.class);
    String preamble = tuple.get("preamble", String.class);
}
```

The names of the `Tuple` elements are determined by the aliases given to the projected items in the select list. If no aliases are specified, the elements may be accessed by their position in the list, where the first item is assigned the position zero.

As an extension to JPA, and in a similar vein, Hibernate lets us pass `Map` or `List`, and have each result packaged as a map or list:

```
var results =
        entityManager.createQuery("select title as title, left(text, 200) as preamble from Book",
                                  Map.class)
            .getResultList();
for (var map : results) {
    String title = (String) map.get("title");
    String preamble = (String) map.get("preamble");
}
```

```
var results =
        entityManager.createQuery("select title, left(text, 200) from Book",
                                  List.class)
            .getResultList();
for (var list : results) {
    String title = (String) list.get(0);
    String preamble = (String) list.get(1);
}
```

Unfortunately, not one of the types `Object[]`, `List`, `Map`, nor `Tuple` lets us access an individual item in a result tuple without a type cast. Sure, `Tuple` does the type cast for us when we pass a class object to `get()`, but it's logically identical. Fortunately there's one more option, as we're about to see.

Actually, `Tuple` really exists to service the criteria query API, and in that context it *does* enable truly typesafe access to query results.

Hibernate 6 lets us pass an arbitrary class type with an appropriate constructor to `createQuery()` and will use it to package the query results. This works extremely nicely with `record` types.

```
record BookSummary(String title, String summary) {}

List<BookSummary> results =
        entityManager.createQuery("select title, left(text, 200) from Book",
                                  BookSummary.class)
            .getResultList();
for (var result : results) {
    String title = result.title();
    String preamble = result.summary();
}
```

It's important that the constructor of `BookSummary` has parameters which exactly match the items in the `select` list.

This class does not need to be mapped or annotated in any way.

Even if the class *is* an entity class, the resulting instances are *not* managed entities and are *not* associated with the session.

We must caution that this still isn't typesafe. In fact, we've just pushed the typecasts down into the call to `createQuery()`. But at least we don't have to write them explicitly.

1.6.3. Instantiation

In JPQL, and in older versions of Hibernate, this functionality required more ceremony.

Result type	Legacy syntax	Streamlined syntax	JPA standard
`Map`	`select new map(x, y)`	`select x, y`	✗/✗
`List`	`select new list(x, y)`	`select x, y`	✗/✗
Arbitrary class `Record`	`select new Record(x, y)`	`select x, y`	✓/✗

For example, the JPA-standard `select new` construct packages the query results into a user-written Java class instead of an array.

```
record BookSummary(String title, String summary) {}

List<BookSummary> results =
        entityManager.createQuery("select new BookSummary(title, left(text, 200)) from Book",
                                  BookSummary.class)
            .getResultList();
for (var result : results) {
    String title = result.title();
    String preamble = result.summary();
}
```

Simplifying slightly, the BNF for a projected item is:

```
selection
    : (expression | instantiation) alias?

instantiation
    : "NEW" instantiationTarget "(" selection ("," selection)* ")"

alias
    : "AS"? identifier
```

Where the list of `selection`s in an `instantiation` is essentially a nested projection list.

Chapter 2. Expressions

We now switch gears, and begin describing the language from the bottom up. The very bottom of a programming language is its syntax for literal values.

2.1. Literals

The most important literal value in this language is `null`. It's assignable to any other type.

2.1.1. Boolean literals

The boolean literal values are the (case-insensitive) keywords `true` and `false`.

2.1.2. String literals

String literals are enclosed in single quotes.

```
select 'hello world'
```

To escape a single quote within a string literal, use a doubled single quote: `''`.

```
from Book where title like 'Ender''s'
```

Alternatively, Java-style double-quoted strings are also allowed, with the usual Java character escape syntax.

```
select "hello\tworld"
```

This option is not much used.

2.1.3. Numeric literals

Numeric literals come in several different forms:

Kind	Type	Example
Integer literals	`Long`, `Integer`, `BigInteger`	`1`, `3_000_000L`, `2BI`
Decimal literals	`Double`, `Float`, `BigDecimal`	`1.0`, `123.456F`, `3.14159265BD`
Hexadecimal literals	`Long`, `Integer`	`0X1A2B`, `0x1a2b`
Scientific notation	`Double`, `Float`, `BigDecimal`	`1e-6`, `6.674E-11F`

For example:

```
from Book where price < 100.0
```

```
select author, count(book)
from Author as author
    join author.books as book
group by author
having count(book) > 10
```

The type of a numeric literal may be specified using a Java-style postfix:

Postfix	Type	Java type
`L` or `l`	long integer	`long`
`D` or `d`	double precision	`double`
`F` or `f`	single precision	`float`

Postfix	Type	Java type
`BI` or `bi`	large integer	`BigInteger`
`BD` or `bd`	exact decimal	`BigDecimal`

It's not usually necessary to specify the precision explicitly.

In a literal with an exponent, the `E` is case-insensitive. Similarly, the Java-style postfix is case-insensitive.

2.1.4. Date and time literals

According to the JPQL specification, date and time literals may be specified using the JDBC escape syntax. Since this syntax is rather unpleasant to look at, HQL provides not one, but two alternatives.

Date/time type	Recommended Java type	JDBC escape syntax 💀	Braced literal syntax	Explicitly typed literal syntax
Date	`LocalDate`	`{d 'yyyy-mm-dd'}`	`{yyyy-mm-dd}`	`date yyyy-mm-dd`
Time	`LocalTime`	`{t 'hh:mm'}`	`{hh:mm}`	`time hh:mm`
Time with seconds	`LocalTime`	`{t 'hh:mm:ss'}`	`{hh:mm:ss}`	`time hh:mm:ss`
Datetime	`LocalDateTime`	`{ts 'yyyy-mm-ddThh:mm:ss'}`	`{yyyy-mm-dd hh:mm:ss}`	`datetime yyyy-mm-dd hh:mm:ss`
Datetime with milliseconds	`LocalDateTime`	`{ts 'yyyy-mm-ddThh:mm:ss.millis'}`	`{yyyy-mm-dd hh:mm:ss.millis}`	`datetime yyyy-mm-dd hh:mm:ss.millis`
Datetime with an offset	`OffsetDateTime`	`{ts 'yyyy-mm-ddThh:mm:ss+hh:mm'}`	`{yyyy-mm-dd hh:mm:ss +hh:mm}`	`datetime yyyy-mm-dd hh:mm:ss +hh:mm`
Datetime with a time zone	`OffsetDateTime`	`{ts 'yyyy-mm-ddThh:mm:ss GMT'}`	`{yyyy-mm-dd hh:mm:ss GMT}`	`datetime yyyy-mm-dd hh:mm:ss GMT`

Literals referring to the current date and time are also provided. Again there is some flexibility.

Date/time type	Java type	Underscored syntax	Spaced syntax
Date	`java.time.LocalDate`	`local_date`	`local date`
Time	`java.time.LocalTime`	`local_time`	`local time`
Datetime	`java.time.LocalDateTime`	`local_datetime`	`local datetime`
Offset datetime	`java.time.OffsetDateTime`	`offset_datetime`	`offset datetime`
Instant	`java.time.Instant`	`instant`	`instant`
Date	`java.sql.Date` 💀	`current_date`	`current date`
Time	`java.sql.Time` 💀	`current_time`	`current time`
Datetime	`java.sql.Timestamp` 💀	`current_timestamp`	`current timestamp`

Of these, only `local date`, `local time`, `local datetime`, `current_date`, `current_time`, and `current_timestamp` are defined by the JPQL specification.

The use of date and time types from the `java.sql` package is strongly discouraged! Always use `java.time` types in new code.

2.1.5. Duration literals

There are two sorts of duration in HQL:

- *year-day durations*, that is, the length of an interval between two dates, and
- *week-nanosecond durations*, that is, the length of an interval between two datetimes.

For conceptual reasons, the two kinds of duration cannot be cleanly composed.

Literal duration expressions are of form `n unit`, for example `1 day` or `10 year` or `100 nanosecond`.

```
select start, end, start - 30 minute from Event
```

The unit may be: `day`, `week`, `month`, `quarter`, `year`, `second`, `minute`, `hour`, or `nanosecond`.

A HQL duration is considered to map to a Java `java.time.Duration`, but semantically they're perhaps more similar to an ANSI SQL `INTERVAL` type.

2.1.6. Binary string literals

HQL also provides a choice of formats for binary strings:

- the braced syntax `{0xDE, 0xAD, 0xBE, 0xEF}`, a list of Java-style hexadecimal byte literals, or
- the quoted syntax `X'DEADBEEF'` or `x'deadbeef'`, similar to SQL.

2.1.7. Enum literals

Literal values of a Java enumerated type may be written without needing to specify the enum class name:

```
from Book where status <> OUT_OF_PRINT
```

```
from Book where type in (BOOK, MAGAZINE)
```

```
update Book set status = OUT_OF_PRINT
```

In the examples above, the enum class is inferred from the type of the expression on the left of the comparison, assignment operator, or `in` predicate.

```
from Book
order by
    case type
    when BOOK then 1
    when MAGAZINE then 2
    when JOURNAL then 3
    else 4
    end
```

In this example the enum class is inferred from the type of the `case` expression.

2.1.8. Java constants

HQL allows any Java `static` constant to be used in HQL, but it must be referenced by its fully-qualified name:

```
select java.lang.Math.PI
```

2.1.9. Literal entity names

Entity names may also occur as a literal value. They do not need to be qualified.

```
from Payment as payment
where type(payment) = CreditCardPayment
```

See Types and typecasts.

2.2. Identification variables and path expressions

A path expression is either:

- a reference to an identification variable, or
- a *compound path*, beginning with a reference to an identification variable, and followed by a period-separated list of references to entity attributes.

As an extension to the JPA spec, HQL, just like SQL, allows a compound path expression where the identification variable at the beginning of the path is missing. That is, instead of `var.foo.bar`, it's legal to write just `foo.bar`. But this is only allowed when the identification variable may be unambiguously inferred from the first element, `foo` of the compound path. The query must have exactly one identification variable `var` for which the path `var.foo` refers to an entity attribute. Note that we will continue to call these paths "compound", even if they only have one element.

This streamlines the query rather nicely when there's just one root entity and no joins. But when the query has multiple identification variables it makes the query much harder to understand.

If an element of a compound path refers to an association, the path expression produces an implicit join.

```
select book.publisher.name from Book book
```

An element of a compound path referring to a many-to-one or on-to-one association may have the treat function applied to it.

```
select treat(order.payment as CreditCardPayment).cardNumber from Order order
```

If an element of a compound path refers to a collection or many-valued association, it must have one of these special functions applied to it.

```
select element(book.authors).name from Book book
```

No other function may be applied to a non-terminal element of a path expression.

Alternatively, if the element of the compound path refers to a list or map, it may have the indexing operator applied to it:

```
select book.editions[0].date from Book book
```

No other operator may be applied to a non-terminal element of a path expression.

2.3. Operator expressions

HQL has operators for working with strings, numeric values, and date/time types.

The operator precedence is given by this table, from highest to lowest precedence:

Precedence class	Type	Operators
Grouping and tuple instantiation		`( … )` and `(x, y, z)`
Case lists		`case … end`
Member reference	Binary infix	`a.b`
Function application	Postfix	`f(x,y)`
Indexing	Postfix	`a[i]`
Unary numeric	Unary prefix	`+`, `-`
Duration conversions	Unary postfix	`by day` and friends
Binary multiplicative	Binary infix	`*`, `/`, `%`
Binary additive	Binary infix	`+`, `-`
Concatenation	Binary infix	`||`

Precedence class	Type	Operators
Nullness, emptiness, truth	Unary postfix	`is null`, `is empty`, `is true`, `is false`
Containment	Binary infix	`in`, `not in`
Between	Ternary infix	`between`, `not between`
Pattern matching	Binary infix	`like`, `ilike`, `not like`, `not ilike`
Comparison operators	Binary infix	`=`, `<>`, `<`, `>`, `<=`, `>=`
Nullsafe comparison	Binary infix	`is distinct from`, `is not distinct from`
Existence	Unary prefix	`exists`
Membership	Binary infix	`member of`, `not member of`
Logical negation	Unary prefix	`not`
Logical conjunction	Binary infix	`and`
Logical disjunction	Binary infix	`or`

2.3.1. String concatenation

JPQL defines two ways to concatenate strings:

- the SQL-style concatenation operator, `||`, and
- the standard `concat()` function.

See below for details of the `concat()` function.

```
select book.title || ' by ' || listagg(author.name, ' & ')
from Book as book
    join book.authors as author
group by book
```

Many more operations on strings are defined below, in Functions.

2.3.2. Numeric arithmetic

The basic SQL arithmetic operators, `+`,`-`,`*`, and `/` are joined by the remainder operator `%`.

```
select (1.0 + :taxRate) * sum(item.book.price * item.quantity)
from Order as ord
    join ord.items as item
where ord.id = :oid
```

When both operands of a binary numeric operator have the same type, the result type of the whole expression is the same as the operands.

By default, the semantics of integer division depend on the database:

- On most databases, division of an integer by an integer evaluates to an integer, just like in Java. Thus, `3/2` evaluates to `1`.
- But on some databases, including Oracle, MySQL, and MariaDB, integer division may result in a non-integral value. So `3/2` evaluates to `1.5` on these databases.

This default behavior may be changed using configuration property `hibernate.query.hql.portable_integer_division`. Setting this property to `true` instructs Hibernate to produce SQL that emulates Java-style integer division (that is, `3/2 = 1`) on platforms where that is not the native semantics.

When the operands are of different types, one of the operands is implicitly converted to *wider* type, with wideness given, in decreasing order, by the list below:

- `Double` (widest)
- `Float`
- `BigDecimal`
- `BigInteger`
- `Long`
- `Integer`
- `Short`
- `Byte`

Many more numeric operations are defined below, in Functions.

2.3.3. Datetime arithmetic

Arithmetic involving dates, datetimes, and durations is quite subtle. Among the issues to consider are:

- There are two kinds of duration: year-day, and week-nanosecond durations. The first is a difference between dates; the second is a difference between datetimes.
- We can subtract dates and datetimes, but we can't add them.
- A Java-style duration has much too much precision, and so in order to use it for anything useful, we must somehow truncate it to something coarser-grained.

Here we list the basic operations.

Operator	Expression type	Example	Resulting type
`-`	Difference between two dates	`your.birthday - local date`	year-day duration
`-`	Difference between two datetimes	`local datetime - record.lastUpdated`	week-nanosecond duration
`-`	Difference of a date and a year-day duration	`local date - 1 day`	date
`-`	Difference of a datetime and a week-nanosecond duration	`record.lastUpdated - 1 minute`	datetime
`-`	Difference between two durations	`1 week - 1 day`	duration
`+`	Sum of a date and a year-day duration	`local date + 1 week`	date
`+`	Sum of a datetime and a week-nanosecond duration	`record.lastUpdated + 1 second`	datetime
`+`	Sum of two durations	`1 day + 4 hour`	duration
`*`	Product of an integer and a duration	`billing.cycles * 30 day`	duration
`by unit`	Convert a duration to an integer	`(1 year) by day`	integer

The `by unit` operator converts a duration to an integer, for example: `(local date - your.birthday) by day` evaluates to the number of days you still have to wait.

The function `extract(unit from …)` extracts a field from a date, time, or datetime type, for example, `extract(year from your.birthday)` produces the year in which you were born, and throws away important information about your birthday.

Please carefully note the difference between these two operations: `by` and `extract()` both evaluate to an integer, but they have very different uses.

Additional datetime operations, including the useful `format()` function, are defined below, in Functions.

2.4. Case expressions

Just like in standard SQL, there are two forms of case expression:

- the *simple* case expression, and
- the so-called *searched* case expression.

Case expressions are verbose. It's often simpler to use the `coalesce()`, `nullif()`, or `ifnull()` functions, as described below in Functions for working with null values.

Simple case expressions

The syntax of the simple form is defined by:

```
"CASE" expression ("WHEN" expression "THEN" expression)+ ("ELSE" expression)? "END"
```

For example:

```
select
    case author.nomDePlume
        when '' then person.name
        else author.nomDePlume end
from Author as author
    join author.person as person
```

Searched case expressions

The searched form has the following syntax:

```
"CASE" ("WHEN" predicate "THEN" expression)+ ("ELSE" expression)? "END"
```

For example:

```
select
    case
        when author.nomDePlume is null then person.name
        else author.nomDePlume end
from Author as author
    join author.person as person
```

A `case` expression may contain complex expression, including operator expressions.

2.5. Tuples

A *tuple instantiation* is an expression like `(1, 'hello')`, and may be used to "vectorize" comparison expressions.

```
from Person where (firstName, lastName) = ('Ludwig', 'Boltzmann')
```

```
from Event where (year, day) > (year(local date), day(local date))
```

This syntax may be used even when the underlying SQL dialect does *not* support so-called "row value" constructors.

A tuple value may be compared to an embedded field:

```
from Person
where address = ('1600 Pennsylvania Avenue, NW', 'Washington', 'DC', 20500, 'USA')
```

2.6. Functions

Both HQL and JPQL define some standard functions and make them portable between databases.

A program that wishes to remain portable between Jakarta Persistence providers should in principle limit itself to the use of the functions which are blessed by the specification. Unfortunately, there's not so many of them.

In some cases, the syntax of these functions looks a bit funny at first, for example, `cast(number as String)`, or `extract(year from date)`, or even `trim(leading '.' from string)`. This syntax is inspired by standard ANSI SQL, and we promise you'll get used to it.

HQL abstracts away from the actual database-native SQL functions, letting you write queries which are portable between databases.

For some functions, and always depending on the database, a HQL function invocation translates to a quite complicated SQL expression!

In addition, there are several ways to use a database function that's not known to Hibernate.

2.6.1. Types and typecasts

The following special functions make it possible to discover or narrow expression types:

Special function	Purpose	Signature	JPA standard
`type()`	The (concrete) entity or embeddable type	`type(e)`	✔
`treat()`	Narrow an entity or embeddable type	`treat(e as Entity)`	✔
`cast()`	Narrow a basic type	`cast(x as Type)`	✔
`str()`	Cast to a string	`str(x)`	✖
`ordinal()`	Get the ordinal value of an enum	`ordinal(x)`	✖

Let's see what these functions do.

Evaluating an entity type

The function `type()`, applied to an identification variable or to an entity-valued or embeddable-valued path expression, evaluates to the concrete type, that is, the Java `Class`, of the referenced entity or embeddable. This is mainly useful when dealing with entity inheritance hierarchies.

```
select payment
from Payment as payment
where type(payment) = CreditCardPayment
```

Narrowing an entity type

The function `treat()` may be used to narrow the type of an identification variable. This is useful when dealing with entity or embeddable inheritance hierarchies.

```
select payment
from Payment as payment
where length(treat(payment as CreditCardPayment).cardNumber)
        between 16 and 20
```

The type of the expression `treat(p as CreditCardPayment)` is the narrowed type, `CreditCardPayment`, instead of the declared type `Payment` of `p`. This allows the attribute `cardNumber` declared by the subtype `CreditCardPayment` to be referenced.

- The first argument is usually an identification variable.
- The second argument is the target type given as an unqualified entity name.

The `treat()` function may even occur in a join.

General typecasts

The function `cast()` has a similar syntax, but is used to narrow basic types.

- Its first argument is usually an attribute of an entity, or a more complex expression involving entity attributes.
- Its second argument is the target type given as an unqualified Java class name: `String`, `Long`, `Integer`, `Double`, `Float`, `Character`, `Byte`, `BigInteger`, `BigDecimal`, `LocalDate`, `LocalTime`, `LocalDateTime`, etc.

```
select cast(id as String) from Order
```

Casting to string

The function `str(x)` is a synonym for `cast(x as String)`.

```
select str(id) from Order
```

Extracting the ordinal value of an enum

The function `ordinal(x)` extracts the ordinal value of an enum. It supports both enum fields mapped as `ORDINAL` and `STRING`.

```
select ordinal(p.type) from Phone p
```

2.6.2. Functions for working with null values

The following functions make it easy to deal with null values:

Function	Purpose	Signature	JPA standard
`coalesce()`	First non-null argument	`coalesce(x, y, z)`	✔
`ifnull()`	Second argument if first is null	`ifnull(x,y)`	✖
`nullif()`	`null` if arguments are equal	`nullif(x,y)`	✔

Handling null values

The `coalesce()` function is a sort of abbreviated `case` expression that returns the first non-null operand.

```
select coalesce(author.nomDePlume, person.name)
from Author as author
    join author.person as person
```

Handling null values

HQL allows `ifnull()` as a synonym for `coalesce()` in the case of exactly two arguments.

```
select ifnull(author.nomDePlume, person.name)
from Author as author
    join author.person as person
```

Producing null values

On the other hand, `nullif()` evaluates to null if its operands are equal, or to its first argument otherwise.

```
select ifnull(nullif(author.nomDePlume, person.name), 'Real name')
from Author as author
    join author.person as person
```

2.6.3. Functions for working with dates and times

There are some very important functions for working with dates and times.

Special function	Purpose	Signature	JPA standard
`extract()`	Extract a datetime field	`extract(field from x)`	✔
`format()`	Format a datetime as a string	`format(datetime as pattern)`	✖
`trunc()` or `truncate()`	Datetime truncation	`truncate(datetime, field)`	✖

Extracting date and time fields

The special function `extract()` obtains a single field of a date, time, or datetime.

- Its first argument is an expression that evaluates to a date, time, or datetime.
- Its second argument is a date/time *field type*.

Recognized Field types are listed below.

Field	Type	Range	Notes	JPA standard
`day`	`Integer`	1-31	Calendar day of month	✔
`month`	`Integer`	1-12		✔
`year`	`Integer`			✔
`week`	`Integer`	1-53	ISO-8601 week number (different to `week of year`)	✔
`quarter`	`Integer`	1-4	Quarter defined as 3 months	✔
`hour`	`Integer`	0-23	Standard 24-hour time	✔
`minute`	`Integer`	0-59		✔
`second`	`Float`	0-59	Includes fractional seconds	✔
`nanosecond`	`Long`		Granularity varies by database	✖
`day of week`	`Integer`	1-7		✖
`day of month`	`Integer`	1-31	Synonym for `day`	✖
`day of year`	`Integer`	1-365		✖
`week of month`	`Integer`	1-5		✖
`week of year`	`Integer`	1-53		✖
`epoch`	`Long`		Elapsed seconds since January 1, 1970	✖
`date`	`LocalDate`		Date part of a datetime	✔
`time`	`LocalTime`		Time part of a datetime	✔
`offset`	`ZoneOffset`		Timezone offset	✖
`offset hour`	`Integer`		Hours of offset	✖
`offset minute`	`Integer`	0-59	Minutes of offset	✖

For a full list of field types, see the Javadoc for `TemporalUnit`.

```
from Order where extract(date from created) = local date
```

```
select extract(year from created), extract(month from created) from Order
```

The following functions are abbreviations for `extract()`:

Function	Long form using `extract()`	JPA standard
`year(x)`	`extract(year from x)`	✖

Function	Long form using `extract()`	JPA standard
`month(x)`	`extract(month from x)`	✗
`day(x)`	`extract(day from x)`	✗
`hour(x)`	`extract(hour from x)`	✗
`minute(x)`	`extract(minute from x)`	✗
`second(x)`	`extract(second from x)`	✗

These abbreviations aren't part of the JPQL standard, but on the other hand they're a lot less verbose.

```
select year(created), month(created) from Order
```

Formatting dates and times

The `format()` function formats a date, time, or datetime according to a pattern.

- Its first argument is an expression that evaluates to a date, time, or datetime.
- Its second argument is a formatting pattern, given as a string.

The pattern must be written in a subset of the pattern language defined by Java's `java.time.format.DateTimeFormatter`.

```
select format(local datetime as 'yyyy-MM-dd HH:mm:ss')
```

For a full list of `format()` pattern elements, see the Javadoc for `Dialect.appendDatetimeFormat`.

Truncating a date or time type

The `truncate()` function truncates the precision of a date, time, or datetime to the temporal unit specified by field type.

- Its first argument is an expression that evaluates to a date, time, or datetime.
- Its second argument is a date/time field type, specifying the precision of the truncated value.

Supported temporal units are: `year`, `month`, `day`, `hour`, `minute` or `second`.

```
select trunc(local datetime, hour)
```

Truncating a date, time or datetime value means obtaining a value of the same type in which all temporal units smaller than `field` have been pruned. For hours, minutes, and seconds this means setting them to `00`. For months and days, this means setting them to `01`.

2.6.4. Functions for working with strings

Naturally, there are a good number of functions for working with strings.

Function	Purpose	Syntax	JPA standard / ANSI SQL Standard
`upper()`	The string, with lowercase characters converted to uppercase	`upper(str)`	✔ / ✔
`lower()`	The string, with uppercase characters converted to lowercase	`lower(str)`	✔ / ✔
`length()`	The length of the string	`length(str)`	✔ / ✗
`concat()`	Concatenate strings	`concat(x, y, z)`	✔ / ✗

Function	Purpose	Syntax	JPA standard / ANSI SQL Standard
`locate()`	Location of string within a string	`locate(patt, str)`, `locate(patt, str, start)`	✔ / ✖
`position()`	Similar to `locate()`	`position(patt in str)`	✖ / ✔
`substring()`	Substring of a string (JPQL-style)	`substring(str, start)`, `substring(str, start, len)`	✔ / ✖
`substring()`	Substring of a string (ANSI SQL-style)	`substring(str from start)`, `substring(str from start for len)`	✖ / ✔
`trim()`	Trim characters from string	`trim(str)`, `trim(leading from str)`, `trim(trailing from str)`, or `trim(leading char from str)`	✔ / ✔
`overlay()`	For replacing a substring	`overlay(str placing rep from start)`, `overlay(str placing rep from start for len)`	✖ / ✔
`pad()`	Pads a string with whitespace, or with a specified character	`pad(str with len)`, `pad(str with len leading)`, `pad(str with len trailing)`, or `pad(str with len leading char)`	✖ / ✖
`left()`	The leftmost characters of a string	`left(str, len)`	✔ / ✖
`right()`	The rightmost characters of a string	`right(str, len)`	✔ / ✖
`replace()`	Replace every occurrence of a pattern in a string	`replace(str, patt, rep)`	✔ / ✖
`repeat()`	Concatenate a string with itself multiple times	`repeat(str, times)`	✖ / ✖
`collate()`	Select a collation	`collate(p.name as collation)`	✖ / ✖
`hex()`	Encode a binary value as a hexadecimal string	`hex(image.bytes)`	✖ / ✖

Let's take a closer look at just some of these.

Contrary to Java, positions of characters within strings are indexed from 1 instead of 0!

Concatenating strings

The JPQL-standard and ANSI SQL-standard `concat()` function accepts a variable number of arguments, and produces a string by concatenating them.

```
select concat(book.title, ' by ', listagg(author.name, ' & '))
from Book as book
    join book.authors as author
group by book
```

Finding substrings

The JPQL function `locate()` determines the position of a substring within another string.

- The first argument is the pattern to search for within the second string.
- The second argument is the string to search in.
- The optional third argument is used to specify a position at which to start the search.

```
select locate('Hibernate', title) from Book
```

The `position()` function has a similar purpose, but follows the ANSI SQL syntax.

```
select position('Hibernate' in title) from Book
```

Slicing strings

Unsurprisingly, `substring()` returns a substring of the given string.

- The second argument specifies the position of the first character of the substring.
- The optional third argument specifies the maximum length of the substring.

```
select substring(title, 1, position(' for Dummies' in title)) from Book         /* JPQL-style */

select substring(title from 1 for position(' for Dummies' in title)) from Book  /* ANSI SQL-style */
```

Alternatively, slicing may be performed using an operator, which is just syntax sugar for the `substring()` function:

```
select title[1:position(' for Dummies' in title)] from Book         /* Operator-style */


select name.first[1]||name.last[1] as initials from Author
```

Trimming strings

The `trim()` function follows the syntax and semantics of ANSI SQL. It may be used to trim `leading` characters, `trailing` characters, or both.

```
select trim(title) from Book


select trim(trailing ' ' from text) from Book
```

Its BNF is funky:

```
"TRIM" "(" (("LEADING" | "TRAILING" | "BOTH")? trimCharacter? "FROM")? expression ")" ;
```

Padding strings

The `pad()` function has a syntax inspired by `trim()`.

```
select concat(pad(b.title with 40 trailing '.'),
              pad(a.firstName with 10 leading),
              pad(a.lastName with 10 leading))
from Book as b
    join b.authors as a
```

Its BNF is given by:

```
"PAD" "(" expression "WITH" expression ("LEADING" | "TRAILING") padCharacter? ")"
```

Collations

The `collate()` function selects a collation to be used for its string-valued argument. Collations are useful for binary comparisons with < or >, and in the order by clause.

For example, `collate(p.name as ucs_basic)` specifies the SQL standard collation `ucs_basic`.

Collations aren't very portable between databases.

Some PostgreSQL collation names must be quoted with backticks, for example, `collate(name as `zh_TW.UTF-8`)`.

The `@Collate` annotation may be used to specify the collation of a column, which is usually more convenient than

> using the `collate()` function.

2.6.5. Numeric functions

Of course, we also have a number of functions for working with numeric values.

Function	Purpose	Signature	JPA standard
`abs()`	The magnitude of a number	`abs(x)`	✔
`sign()`	The sign of a number	`sign(x)`	✔
`mod()`	Remainder of integer division	`mod(n,d)`	✔
`sqrt()`	Square root of a number	`sqrt(x)`	✔
`exp()`	Exponential function	`exp(x)`	✔
`power()`	Exponentiation	`power(x,y)`	✔
`ln()`	Natural logarithm	`ln(x)`	✔
`round()`	Numeric rounding	`round(number)`, `round(number, places)`	✔
`trunc()` or `truncate()`	Numeric truncation	`truncate(number)`, `truncate(number, places)`	✘
`floor()`	Floor function	`floor(x)`	✔
`ceiling()`	Ceiling function	`ceiling(x)`	✔
`log10()`	Base-10 logarithm	`log10(x)`	✘
`log()`	Arbitrary-base logarithm	`log(b,x)`	✘
`pi`	π	`pi`	✘
`sin()`, `cos()`, `tan()`, `asin()`, `acos()`, `atan()`	Basic trigonometric functions	`sin(theta)`, `cos(theta)`	✘
`atan2()`	Two-argument arctangent (range `(-π,π]`)	`atan2(y, x)`	✘
`sinh()`, `cosh()`, `tanh()`	Hyperbolic functions	`sinh(x)`, `cosh(x)`, `tanh(x)`	✘
`degrees()`	Convert radians to degrees	`degrees(x)`	✘
`radians()`	Convert degrees to radians	`radians(x)`	✘
`least()`	Return the smallest of the given arguments	`least(x, y, z)`	✘
`greatest()`	Return the largest of the given arguments	`greatest(x, y, z)`	✘
`bitand()`, `bitor()`, `bitxor()`	Bitwise functions	`bitand(x,y)`	✘

We haven't included aggregate functions, ordered set aggregate functions, or window functions in this list, because their purpose is more specialized, and because they come with extra special syntax.

2.6.6. Functions for dealing with collections

The functions described in this section are especially useful when dealing with `@ElementCollection` mappings, or with collection mappings involving an `@OrderColumn` or `@MapKeyColumn`.

The following functions accept either:

1. an identification variable that refers to a joined collection or many-valued association, or
2. a compound path that refers to a collection or many-valued association of an entity.

In case 2, application of the function produces an implicit join.

Function	Applies to	Purpose	JPA standard
`size()`	Any collection	The size of a collection	✔
`element()`	Any collection	The element of a set or list	✖
`index()`	Lists	The index of a list element	✔
`key()`	Maps	The key of a map entry	✔
`value()`	Maps	The value of a map entry	✔
`entry()` 💀	Maps	The whole entry in a map	✔

The next group of functions always accept a compound path referring to a collection or many-valued association of an entity. They're interpreted as referring to the collection as a whole.

Function	Applies to	Purpose	JPA standard
`elements()`	Any collection	The elements of a set or list, collectively	✖
`indices()`	Lists	The indexes of a list, collectively	✖
`keys()`	Maps	The keys of a map, collectively	✖
`values()`	Maps	The values of a map, collectively	✖

Application of one of these functions produces an implicit subquery or implicit join.

This query has an implicit join:

```
select title, element(tags) from Book
```

This query has an implicit subquery:

```
select title from Book where 'hibernate' in elements(tags)
```

It never makes sense to apply the functions `elements()`, `indices()`, `keys()`, or `values()` to an identification variable or single-valued path expression. These functions must be applied to a reference to a many-valued path expression.

Collection sizes

The `size()` function returns the number of elements of a collection or to-many association.

```
select name, size(books) from Author
```

Set or list elements

The `element()` function returns a reference to an element of a joined set or list. For an identification variable (case 1 above), this function is optional. For a compound path (case 2), it's required.

List indexes

The `index()` function returns a reference to the index of a joined list.

In this example, `element()` is optional, but `index()` is required:

```
select id(book), index(ed), element(ed)
from Book book as book
    join book.editions as ed
```

Map keys and values

The `key()` function returns a reference to a key of a joined map. The `value()` function returns a reference to its value.

```
select key(entry), value(entry)
from Thing as thing
    join thing.entries as entry
```

Quantification over collections

The functions `elements()`, `indices()`, `keys()`, and `values()` are used to quantify over collections. We may use them with:

- an `in` or `exists` predicate,
- a relational comparison, or
- an aggregate function.

Shortcut	Equivalent subquery
`exists elements(book.editions)`	`exists (select ed from book.editions as ed)`
`2 in indices(book.editions)`	`2 in (select index(ed) from book.editions as ed)`
`10 > all(elements(book.printings))`	`10 > all(select pr from book.printings as pr)`
`max(elements(book.printings))`	`(select max(pr) from book.printings as pr)`

For example:

```
select title from Book where 'hibernate' in elements(tags)
```

Don't confuse the `elements()` function with `element()`, the `indices()` function with `index()`, the `keys()` function with `key()`, or the `values()` function with `value()`. The functions named in singular deal with elements of "flattened" collections. If not already joined, they add an implicit join to the query. The functions with plural naming do *not* flatten a collection by joining it.

The following queries are different:

```
select title, max(index(revisions)) from Book  /* implicit join */
```

```
select title, max(indices(revisions)) from Book  /* implicit subquery */
```

The first query produces a single row, with `max()` taken over all books. The second query produces a row per book, with `max()` taken over the collection elements belonging to the given book.

2.6.7. Functions for working with ids and versions

Finally, the following functions evaluate the id, version, or natural id of an entity, or the foreign key of a to-one association:

Function	Purpose	JPA standard
`id()`	The value of the entity `@Id` attribute.	✔
`version()`	The value of the entity `@Version` attribute.	✔

Function	Purpose	JPA standard
`naturalid()`	The value of the entity `@NaturalId` attribute.	✗
`fk()`	The value of the foreign key column mapped by a `@ManyToOne` (or logical `@OneToOne`) association. Useful with associations annotated `@NotFound`.	✗

2.6.8. Array, XML, and JSON functions

On supported platforms, HQL provides a rich suite of functions for working with:

- SQL arrays,
- JSON, and
- XML

The use of these functions is outside the scope of this guide. However, we note that the following language constructs work with arrays, and are implemented as syntactic sugar for the underlying functions:

Syntax	Interpretation
`[1, 2]`	Instantiate an array
`array[1]`	Array element
`array[1:2]`	Array slice
`length(array)`	Length of an array
`position(element in array)`	Position of an element within an array
`cast(array as String)`	Typecast array to string
`element in array` or `array contains element`	Determine if an element belongs to an array
`array includes subarray`	Determine if the elements of one array include all the elements of a second array

2.7. Hash functions

The following functions work on most supported platforms:

Function	Purpose	JPA standard
`sha()`	The SHA256 hash of a string.	✗
`md5()`	The MD5 hash of a string.	✗

These functions accept a string and return `byte[]`. The return value is compatible with the byte array produced by Java's `MessageDigest`.

2.7.1. Embedding SQL expressions

The following special functions let us embed a call to a native SQL function, refer directly to a column, or evaluate an expression written in native SQL.

Function	Purpose	Signature	JPA standard
`function()`	Call a SQL function	`function('fun', arg1, arg2)`	✓
`function()`	Call a SQL function	`function(fun, arg1, arg2)`, `function(fun as Type, arg1, arg2)`	✗

Function	Purpose	Signature	JPA standard
`column()`	A column value	`column(entity.column)`, `column(entity.column as Type)`	✗
`sql()`	Evaluate a SQL expression	`sql('text', arg1, arg2)`	✗

Before using one of these functions, ask yourself if it might be better to just write the whole query in native SQL.

Direct column references

The `column()` function lets us refer to an unmapped column of a table. The column name must be qualified by an identification variable or path expression.

```
select column(log.ctid as String)
from Log log
```

Of course, the table itself must be mapped by an entity class.

Native and user-defined functions

The functions we've described above are the functions abstracted by HQL and made portable across databases. But, of course, HQL can't abstract every function in your database.

There are several ways to call native or user-defined SQL functions.

- A native or user-defined function may be called using JPQL's `function` syntax, for example, `function('sinh', phi)`, or HQL's extension to that syntax, for example `function(sinh as Double, phi)`. (This is the easiest way, but not the best way.)
- A user-written `FunctionContributor` may register user-defined functions.
- A custom `Dialect` may register additional native functions by overriding `initializeFunctionRegistry()`.

Registering a function isn't hard, but is beyond the scope of this guide.

(It's even possible to use the APIs Hibernate provides to make your own *portable* functions!)

Fortunately, every built-in `Dialect` already registers many native functions for the database it supports.

Try setting the log category `org.hibernate.HQL_FUNCTIONS` to debug. Then at startup Hibernate will log a list of type signatures of all registered functions.

Embedding native SQL in HQL

The special function `sql()` allows the use of native SQL fragments inside an HQL query.

The signature of this function is `sql(pattern[, argN]*)`, where `pattern` must be a string literal but the remaining arguments may be of any type. The pattern literal is unquoted and embedded in the generated SQL. Occurrences of `?` in the pattern are replaced with the remaining arguments of the function.

We may use this, for example, to perform a native PostgreSQL typecast:

```
from Computer c where c.ipAddress = sql('?::inet', '127.0.0.1')
```

This results in SQL logically equivalent to:

```
select * from Computer c where c.ipAddress = '127.0.0.1'::inet
```

Or we can use a native SQL operator:

```
from Human h order by sql('(? <-> ?)', h.workLocation, h.homeLocation)
```

And this time the SQL is logically equivalent to:

```
select * from Human h where (h.workLocation <-> h.homeLocation)
```

2.8. Predicates

A predicate is an operator which, when applied to some argument, evaluates to `true` or `false`. In the world of SQL-style ternary logic, we must expand this definition to encompass the possibility that the predicate evaluates to `null`. Typically, a predicate evaluates to `null` when one of its arguments is `null`.

Predicates occur in the `where` clause, the `having` clause and in searched case expressions.

2.8.1. Comparison operators

The binary comparison operators are borrowed from SQL: =, >, >=, <, <=, <>.

If you prefer, HQL treats `!=` as a synonym for <>.

The operands should be of the same type.

```
from Book where price < 1.0
```

```
from Author as author where author.nomDePlume <> author.person.name
```

```
select id, total
from (
    select ord.id as id, sum(item.book.price * item.quantity) as total
    from Order as ord
        join Item as item
    group by ord
)
where total > 100.0
```

2.8.2. The `between` predicate

The ternary `between` operator, and its negation, `not between`, determine if a value falls within a range.

Of course, all three operands must be of compatible type.

```
from Book where price between 1.0 and 100.0
```

2.8.3. Operators for dealing with null

The following operators make it easier to deal with null values. These predicates never evaluate to `null`.

Operator	Negation	Type	Semantics
`is null`	`is not null`	Unary postfix	`true` if the value to the left is null, or false if it is not null
`is distinct from`	`is not distinct from`	Binary	`true` if the value on the left is equal to the value on the right, or if both values are null, and false otherwise

```
from Author where nomDePlume is not null
```

2.8.4. Operators for dealing with boolean values

These operators perform comparisons on values of type `boolean`. These predicates never evaluate to `null`.

The values `true` and `false` of the `boolean` basic type are different to the logical `true` or `false` produced by a predicate.

For *logical* operations on predicates, see Logical operators below.

Operator	Negation	Type	Semantics
`is true`	`is not true`	Unary postfix	`true` if the value to the left is `true`, or `false` otherwise

Operator	Negation	Type	Semantics
`is false`	`is not false`	Binary	`true` if the value to the left is `false`, or `false` otherwise

```
from Book where discontinued is not true
```

2.8.5. Collection predicates

The following operators apply to collection-valued attributes and to-many associations.

Operator	Negation	Type	Semantics
`is empty`	`is not empty`	Unary postfix	`true` if the collection or association on the left has no elements
`member of`	`not member of`	Binary	`true` if the value on the left is a member of the collection or association on the right

```
from Author where books is empty
```

```
select author, book
from Author as author, Book as book
where author member of book.authors
```

2.8.6. String pattern matching

The `like` operator performs pattern matching on strings. Its friend `ilike` performs case-insensitive matching.

Their syntax is defined by:

```
expression "NOT"? ("LIKE" | "ILIKE") expression ("ESCAPE" character)?
```

The expression on the right is a pattern, where:

- _ matches any single character,
- `%` matches any number of characters, and
- if an escape character is specified, it may be used to escape either of these wildcards.

```
from Book where title not like '% for Dummies'
```

The optional `escape` character allows a pattern to include a literal _ or `%` character.

As you can guess, `not like` and `not ilike` are the enemies of `like` and `ilike`, and evaluate to the exact opposite boolean values.

2.8.7. The `in` predicate

The `in` predicates evaluates to true if the value to its left is in … well, whatever it finds to its right.

Its syntax is unexpectedly complicated:

```
expression "NOT"? "IN" inList

inList
    : collectionQuantifier "(" simplePath ")"
    | "(" (expression ("," expression)*)? ")"
    | "(" subquery ")"
    | parameter
    | expression
```

This less-than-lovely fragment of the HQL ANTLR grammar tells us that the thing to the right might be:

- a list of values enclosed in parentheses,
- a subquery,

- one of the collection-handling functions defined above,
- a query parameter, or
- an expression evaluating to a SQL array.

The type of the expression on the left, and the types of all the values on the right must be compatible.

JPQL limits the legal types to string, numeric, date/time, and enum types, and in JPQL the left expression must be either:

- a *state field*, which means a basic attribute, excluding associations and embedded attributes, or
- an *entity type expression*.

HQL is far more permissive. HQL itself does not restrict the type in any way, though the database itself might. Even embedded attributes are allowed, although that feature depends on the level of support for tuple or "row value" constructors in the underlying database.

```
from Payment as payment
where type(payment) in (CreditCardPayment, WireTransferPayment)
```

```
from Author as author
where author.person.name in (select name from OldAuthorData)
```

```
from Book as book
where :edition in elements(book.editions)
```

It's quite common to have a parameterized list of values.

Here's a very useful idiom:

```
List<Book> books =
        session.createSelectionQuery("from Book where isbn in :isbns", Book.class)
            .setParameterList("isbns", listOfIsbns)
            .getResultList();
```

We may even "vectorize" an `in` predicate, using a tuple constructor and a subquery with multiple selection items:

```
from Author as author
where (author.person.name, author.person.birthdate)
    in (select name, birthdate from OldAuthorData)
```

2.8.8. Comparison operators and subqueries

The binary comparisons we met above may involve a quantifier, either:

- a quantified subquery, or
- a quantifier applied to one of the functions defined above.

The quantifiers are unary prefix operators: `all`, `every`, `any`, and `some`.

Subquery operator	Synonym	Semantics
`every`	`all`	Evaluates to true of the comparison is true for *every* value in the result set of the subquery
`any`	`some`	Evaluates to true of the comparison is true for *at least one* value in the result set of the subquery

```
from Publisher pub where 100.0 < all(select price from pub.books)
```

```
from Publisher pub where :title = some(select title from pub.books)
```

2.8.9. The `exists` predicate

The unary prefix `exists` operator evaluates to true if the thing to its right is nonempty.

The thing to its right might be:

- a subquery, or
- one of the functions defined above.

As you can surely guess, `not exists` evaluates to true if the thing to the right *is* empty.

```
from Author where exists elements(books)


from Author as author
where exists (
    from Order join items
    where book in elements(author.books)
)
```

2.8.10. Logical operators

The logical operators are binary infix `and` and `or`, and unary prefix `not`.

Just like SQL, logical expressions are based on ternary logic. A logical operator evaluates to null if it has a null operand.

Chapter 3. Root entities and joins

The `from` clause, and its subordinate `join` clauses sit right at the heart of most queries.

3.1. Declaring root entities

The `from` clause is responsible for declaring the entities available in the rest of the query, and assigning them aliases, or, in the language of the JPQL specification, *identification variables*.

3.1.1. Identification variables

An identification variable is just a name we can use to refer to an entity and its attributes from expressions in the query. It may be any legal Java identifier. According to the JPQL specification, identification variables must be treated as case-insensitive language elements.

The identification variable is actually optional, but for queries involving more than one entity it's almost always a good idea to declare one.

This *works*, but it isn't particularly good form:

```
from Publisher join books join authors join person where ssn = :ssn
```

Identification variables may be declared with the `as` keyword, but this is optional.

3.1.2. Root entity references

A root entity reference, or what the JPQL specification calls a *range variable declaration*, is a direct reference to a mapped `@Entity` type by its entity name.

Remember, the *entity name* is the value of the `name` member of the `@Entity` annotation, or the unqualified Java class name by default.

```
select book from Book as book
```

In this example, `Book` is the entity name, and `book` is the identification variable. The `as` keyword is optional.

Alternatively, a fully-qualified Java class name may be specified. Then Hibernate will query every entity which inherits the named type.

```
select doc from org.hibernate.example.AbstractDocument as doc where doc.text like :pattern
```

Of course, there may be multiple root entities.

```
select a, b
from Author a, Author b, Book book
where a in elements(book.authors)
  and b in elements(book.authors)
```

This query may even be written using the syntax `cross join` in place of the commas:

```
select a, b
from Book book
    cross join Author a
    cross join Author b
where a in elements(book.authors)
  and b in elements(book.authors)
```

Of course, it's possible to write old-fashioned pre-ANSI-era joins:

```
select book.title, publisher.name
from Book book, Publisher publisher
where book.publisher = publisher
  and book.title like :titlePattern
```

But we never write HQL this way.

3.1.3. Polymorphism

HQL and JPQL queries are inherently polymorphic. Consider:

```
select payment from Payment as payment
```

This query names the `Payment` entity explicitly. But the `CreditCardPayment` and `WireTransferPayment` entities inherit `Payment`, and so `payment` ranges over all three types. Instances of all these entities are returned by the query.

The query `from java.lang.Object` is completely legal. (But not very useful!)

It returns every object of every mapped entity type.

3.1.4. Derived roots

A *derived root* is an uncorrelated subquery which occurs in the `from` clause.

```
select id, total
from (
    select ord.id as id, sum(item.book.price * item.quantity) as total
    from Order as ord
        join ord.items as item
    group by ord
)
where total > 100.0
```

The derived root may declare an identification variable.

```
select stuff.id, stuff.total
from (
    select ord.id as id, sum(item.book.price * item.quantity) as total
    from Order as ord
        join ord.items as item
    group by ord
) as stuff
where total > 100.0
```

This feature can be used to break a more complicated query into smaller pieces.

We emphasize that a derived root must be an *uncorrelated* subquery. It may not refer to other roots declared in the same `from` clause.

A subquery may also occur in a join, in which case it may be a correlated subquery.

3.1.5. Common table expressions in `from` clause

A *common table expression (CTE)* is like a derived root with a name. We'll discuss CTEs later.

3.2. Declaring joined entities

Joins allow us to navigate from one entity to another, via its associations, or via explicit join conditions. There are:

- *explicit joins*, declared within the `from` clause using the keyword `join`, and
- *implicit joins*, which don't need to be declared in the `from` clause.

An explicit join may be either:

- an *inner join*, written as `join` or `inner join`,
- a *left outer join*, written as `left join` or `left outer join`,
- a *right outer join*, written as `right join` or `right outer join`, or
- a *full outer join*, written as `full join` or `full outer join`.

3.2.1. Explicit root joins

An explicit root join works just like an ANSI-style join in SQL.

```
select book.title, publisher.name
from Book book
    join Publisher publisher
        on book.publisher = publisher
where book.title like :titlePattern
```

The join condition is written out explicitly in the `on` clause.

This looks nice and familiar, but it's *not* the most common sort of join in HQL or JPQL.

3.2.2. Explicit association joins

Every explicit association join specifies an entity attribute to be joined. The specified attribute:

- is usually a `@OneToMany`, `@ManyToMany`, `@OneToOne`, or `@ManyToOne` association, but
- it could be an `@ElementCollection`, or even just a SQL array, and
- it might even be an attribute of embeddable type.

In the case of an association or collection, the generated SQL will have a join of the same type. (For a many-to-many association it will have *two* joins.) In the case of an embedded attribute, the join is purely logical and does not result in a join in the generated SQL.

An explicit join may assign an identification variable to the joined entity.

```
from Book as book
    join book.publisher as publisher
    join book.authors as author
where book.title like :titlePattern
select book.title, author.name, publisher.name
```

For an outer join, we must write our query to accommodate the possibility that the joined association is missing.

```
from Book as book
    left join book.publisher as publisher
    join book.authors as author
where book.title like :titlePattern
select book.title, author.name, ifnull(publisher.name, '-')
```

For further information about collection-valued association references, see Joining collections and many-valued associations.

3.2.3. Explicit association joins with join conditions

The `with` or `on` clause allows explicit qualification of the join conditions.

The specified join conditions are *added* to the join conditions specified by the foreign key association. That's why, historically, HQL uses the keword `with` here: "with" emphasizes that the new condition doesn't *replace* the original join conditions.

The `with` keyword is specific to Hibernate. JPQL uses `on`.

Join conditions occurring in the `with` or `on` clause are added to the `on` clause in the generated SQL.

```
from Book as book
    left join book.publisher as publisher
        with publisher.closureDate is not null
    left join book.authors as author
        with author.type <> COLLABORATION
where book.title like :titlePattern
select book.title, author.name, publisher.name
```

3.2.4. Association fetching

A *fetch join* overrides the laziness of a given association, specifying that the association should be fetched with a SQL join. The join may be an inner or outer join.

- A `join fetch`, or, more explicitly, `inner join fetch`, only returns base entities with an associated entity.
- A `left join fetch`, or—for lovers of verbosity—`left outer join fetch`, returns all the base entities, including those which have no associated joined entity.

This is one of the most important features of Hibernate. To achieve acceptable performance with HQL, you'll need to use `join fetch` quite often. Without it, you'll quickly run into the dreaded "n+1 selects" problem.

For example, if `Person` has a one-to-many association named `phones`, the use of `join fetch` in the following query specifies that the collection elements should be fetched in the same SQL query:

```
select book
from Book as book
    left join fetch book.publisher
    join fetch book.authors
```

In this example, we used a left outer join for `book.publisher` because we also wanted to obtain books with no publisher, but a regular inner join for `book.authors` because every book has at least one author.

A query may have more than one fetch join, but be aware that:

- it's perfectly safe to fetch several to-one associations in series or parallel in a single query, and
- a single series of *nested* fetch joins is also fine, but
- fetching multiple collections or to-many associations in *parallel* results in a Cartesian product at the database level, and might exhibit very poor performance.

HQL doesn't disallow it, but it's usually a bad idea to apply a restriction to a `join fetch`ed entity, since the elements of the fetched collection would be incomplete. Indeed, it's best to avoid even assigning an identification variable to a fetched joined entity except for the purpose of specifying a nested fetch join.

Fetch joins should usually be avoided in limited or paged queries. This includes:

- queries executed with limits specified via the `setFirstResult()` and `setMaxResults()` methods of `Query`, or
- queries with a limit or offset declared in HQL, described below in Limits and offsets.

Nor should they be used with the `scroll()` and `stream()` methods of the `Query` interface.

Fetch joins are disallowed in subqueries, where they would make no sense.

3.2.5. Joins with typecasts

An explicit join may narrow the type of the joined entity using `treat()`.

```
from Order as ord
    join treat(ord.payments as CreditCardPayment) as ccp
where length(ccp.cardNumber) between 16 and 20
select ord.id, ccp.cardNumber, ccp.amount
```

Here, the identification variable `ccp` declared to the right of `treat()` has the narrowed type `CreditCardPayment`, instead of the declared type `Payment`. This allows the attribute `cardNumber` declared by the subtype `CreditCardPayment` to be referenced in the rest of the query.

See Types and typecasts for more information about `treat()`.

3.2.6. Subqueries in joins

A `join` clause may contain a subquery, either:

- an uncorrelated subquery, which is almost the same as a derived root, except that it may have an `on` restriction, or
- a *lateral join*, which is a correlated subquery, and may refer to other roots declared earlier in the same `from` clause.

The `lateral` keyword just distinguishes the two cases.

```
from Phone as phone
    left join (
        select call.duration as duration, call.phone.id as cid
        from Call as call
        order by call.duration desc
        limit 1
    ) as longest on cid = phone.id
where phone.number = :phoneNumber
select longest.duration
```

This query may also be expressed using a `lateral` join:

```
from Phone as phone
    left join lateral (
       select call.duration as duration
       from phone.calls as call
       order by call.duration desc
       limit 1
    ) as longest
where phone.number = :phoneNumber
select longest.duration
```

A lateral join may be an inner or left outer join, but not a right join, nor a full join.

Traditional SQL doesn't allow correlated subqueries in the `from` clause. A lateral join is essentially just that, but with a different syntax to what you might expect.

On some databases, `join lateral` is written `cross apply`. And on Postgres it's plain `lateral`, without `join`.

It's almost as if they're *deliberately trying* to confuse us.

Lateral joins are particularly useful for computing top-N elements of multiple groups.

Most databases support some flavor of `join lateral`, and Hibernate emulates the feature for databases which don't. But emulation is neither very efficient, nor does it support all possible query shapes, so it's important to test on your target database.

3.2.7. Implicit association joins (path expressions)

It's not necessary to explicitly `join` every entity that occurs in a query. Instead, entity associations may be *navigated*, just like in Java:

- if an attribute is of embedded type, or is a to-one association, it may be further navigated, but
- if an attribute is of basic type, it is considered terminal, and may not be further navigated, and
- if an attribute is collection-valued, or is a to-many association, it may be navigated, but only with the help of `value()`, `element()`, or `key()`.

It's clear that:

- A path expression like `author.name` with only two elements just refers to state held directly by an entity with an alias `author` defined in `from` or `join`.
- But a longer path expression, for example, `author.person.name`, might refer to state held by an associated entity. (Alternatively, it might refer to state held by an embedded class.)

In the second case, Hibernate with automatically add a join to the generated SQL if necessary.

```
from Book as book
where book.publisher.name like :pubName
```

As in this example, implicit joins usually appear outside the `from` clause of the HQL query. However, they always affect the `from` clause of the SQL query.

The example above is equivalent to:

```
select book
from Book as book
    join book.publisher as pub
where pub.name like :pubName
```

Note that:

- Implicit joins are always treated as inner joins.
- Multiple occurrences of the same implicit join always refer to the same SQL join.

This query:

```
select book
from Book as book
where book.publisher.name like :pubName
  and book.publisher.closureDate is null
```

results in just one SQL join, and is just a different way to write:

```
select book
from Book as book
    join book.publisher as pub
where pub.name like :pubName
  and pub.closureDate is null
```

3.2.8. Joining collections and many-valued associations

When a join involves a collection or many-valued association, the declared identification variable refers to the *elements* of the collection, that is:

- to the elements of a `Set`,
- to the elements of a `List`, not to their indices in the list, or
- to the values of a `Map`, not to their keys.

```
select publisher.name, author.name
from Publisher as publisher
    join publisher.books as book
    join book.authors as author
where author.name like :namePattern
```

In this example, the identification variable `author` is of type `Author`, the element type of the list `Book.authors`. But if we need to refer to the index of an `Author` in the list, we need some extra syntax.

You might recall that we mentioned List indexes and Map keys and values a bit earlier. These functions may be applied to the identification variable declared in a collection join or many-valued association join.

Function	Applies to	Interpretation	Notes
`value()` or `element()`	Any collection	The collection element or map entry value	Often optional.
`index()`	Any `List` with an index column	The index of the element in the list	For backward compatibility, it's also an alternative to `key()`, when applied to a map.
`key()`	Any `Map`	The key of the entry in the map	If the key is of entity type, it may be further navigated.
`entry()`	Any `Map`	The map entry, that is, the `Map.Entry` of key and value.	Only legal as a terminal path, and only allowed in the `select` clause.

In particular, `index()` and `key()` obtain a reference to a list index or map key.

```
select book.title, author.name, index(author)
from Book as book
    join book.authors as author
```

```
select publisher.name, leadAuthor.name
from Publisher as publisher
    join publisher.books as book
    join book.authors as leadAuthor
where leadAuthor.name like :namePattern
  and index(leadAuthor) == 0
```

3.2.9. Implicit joins involving collections

A path expression like `book.authors.name` is not considered legal. We can't just navigate a many-valued association with this syntax.

Instead, the functions `element()`, `index()`, `key()`, and `value()` may be applied to a path expression to express an implicit join. So we must write `element(book.authors).name` or `index(book.authors)`.

```
select book.title, element(book.authors).name, index(book.authors)
from Book book
```

An element of an indexed collection (an array, list, or map) may even be identified using the index operator:

```
select publisher.name, book.authors[0].name
from Publisher as publisher
    join publisher.books as book
where book.authors[0].name like :namePattern
```

Chapter 4. Selection, projection, and aggregation

Joining is one kind of *relational operation*. It's an operation that produces relations (tables) from other relations. Such operations, taken together, form the *relational algebra*.

We must now understand the rest of this family: restriction a.k.a. selection, projection, aggregation, union/intersection, and, finally, ordering and limiting, operations which are not strictly part of the calculus of relations, but which usually come along for the ride because they're very *useful*.

We'll start with the operation that's easiest to understand.

4.1. Restriction

The `where` clause restricts the results returned by a `select` query or limits the scope of an `update` or `delete` query.

This operation is usually called *selection*, but since that term is often confused with the `select` keyword, and since both projection and selection involve "selecting" things, here we'll use the less-ambiguous term *restriction*.

A restriction is nothing more than a single logical expression, a topic we exhausted above in Predicates. Therefore, we'll move quickly onto the next, and more interesting, operation.

4.2. Aggregation

An aggregate query is one with aggregate functions in its projection list. It collapses multiple rows into a single row. Aggregate queries are used for summarizing and analysing data.

An aggregate query might have a `group by` clause. The `group by` clause divides the result set into groups, so that a query with aggregate functions in the select list returns not a single result for the whole query, but one result for each group. If an aggregate query *doesn't* have a `group by` clause, it always produces a single row of results.

In short, *grouping* controls the effect of *aggregation*.

A query with aggregation may also have a `having` clause, a restriction applied to the groups.

4.2.1. Aggregation and grouping

The `group by` clause looks quite similar to the `select` clause—it has a list of grouped items, but:

- if there's just one item, then the query will have a single result for each unique value of that item, or
- if there are multiple items, the query will have a result for each unique *combination* or their values.

The BNF for a grouped item is just:

```
identifier | INTEGER_LITERAL | expression
```

Consider the following queries:

```
select book.isbn,
    sum(quantity) as totalSold,
    sum(quantity * book.price) as totalBilled
from Item
where book.isbn = :isbn


select book.isbn,
    year(order.dateTime) as year,
    sum(quantity) as yearlyTotalSold,
    sum(quantity * book.price) as yearlyTotalBilled
from Item
where book.isbn = :isbn
group by year(order.dateTime)
```

The first query calculates complete totals over all orders in years. The second calculates totals for each year, after grouping the orders by year.

4.2.2. Totals and subtotals

The special functions `rollup()` and `cube()` may be used in the `group by` clause, when supported by the database. The semantics are identical to SQL.

These functions are especially useful for reporting.

- A `group by` clause with `rollup()` is used to produce subtotals and grand totals.
- A `group by` clause with `cube()` allows totals for every combination of columns.

4.2.3. Aggregation and restriction

In a grouped query, the `where` clause applies to the non-aggregated values (it determines which rows will make it into the aggregation). The `having` clause also restricts results, but it operates on the aggregated values.

In an example above, we calculated totals for every year for which data was available. But our dataset might extend far back into the past, perhaps even as far back as those terrible dark ages before Hibernate 2.0. So let's restrict our result set to data from our own more civilized times:

```
select book.isbn,
    year(order.dateTime) as year,
    sum(quantity) as yearlyTotalSold,
    sum(quantity * book.price) as yearlyTotalBilled
from Item
where book.isbn = :isbn
group by year(order.dateTime)
having year(order.dateTime) > 2003
   and sum(quantity) > 0
```

The `having` clause follows the same rules as the `where` clause and is also just a logical predicate. The `having` restriction is applied after grouping and aggregation has already been performed, whereas the `where` clause is applied before the data is grouped or aggregated.

4.3. Projection

The `select` list identifies which objects and values to return as the query results. This operation is called *projection*.

```
selectClause
    : "SELECT" "DISTINCT"? selection (","" selection)*
```

Any of the expression types discussed in Expressions may occur in the projection list, unless otherwise noted.

If a query has no explicit `select` list, then, as we saw much earlier, the projection is inferred from the entities and joins occurring in the `from` clause, together with the result type specified by the call to `createQuery()`. But it's better to specify the projection explicitly, except in the simplest cases.

4.3.1. Duplicate removal

The `distinct` keyword helps remove duplicate results from the query result list. Its only effect is to add `distinct` to the generated SQL.

```
select distinct lastName from Person
```

```
select distinct author
from Publisher as pub
    join pub.books as book
    join book.authors as author
where pub.id = :pid
```

As of Hibernate 6, duplicate results arising from the use of `join fetch` are automatically removed by Hibernate in memory, *after* reading the database results and materializing entity instances as Java objects. It's no longer necessary to remove duplicate results explicitly, and, in particular, `distinct` should not be used for this purpose.

4.3.2. Aggregate functions

It's common to have aggregate functions like `count()`, `sum()`, and `max()` in a select list. Aggregate functions are special functions that reduce the size of the result set.

The standard aggregate functions defined in both ANSI SQL and JPQL are these ones:

Aggregate function	Argument type	Result type	JPA standard / ANSI SQL standard
`count()`, including `count(distinct)`, `count(all)`, and `count(*)`	Any	`Long`	✔/✔
`avg()`	Any numeric type	`Double`	✔/✔
`min()`	Any numeric type, or string	Same as the argument type	✔/✔
`max()`	Any numeric type, or string	Same as the argument type	✔/✔
`sum()`	Any numeric type	See table below	✔/✔
`var_pop()`, `var_samp()`	Any numeric type	`Double`	✘/✔
`stddev_pop()`, `stddev_samp()`	Any numeric type	`Double`	✘/✔

```
select count(distinct item.book)
from Item as item
where year(item.order.dateTime) = :year
```

```
select sum(item.quantity) as totalSales
from Item as item
where item.book.isbn = :isbn
```

```
select
    year(item.order.dateTime) as year,
    sum(item.quantity) as yearlyTotal
from Item as item
where item.book.isbn = :isbn
group by year(item.order.dateTime)
```

```
select
    month(item.order.dateTime) as month,
    avg(item.quantity) as monthlyAverage
from Item as item
where item.book.isbn = :isbn
group by month(item.order.dateTime)
```

In the case of `sum()`, the rules for assigning a result type are:

Argument type	Result type
Any integral numeric type except `BigInteger`	`Long`
Any floating point numeric type	`Double`
`BigInteger`	`BigInteger`
`BigDecimal`	`BigDecimal`

HQL defines two additional aggregate functions which accept a logical predicate as an argument.

Aggregate function	Argument type	Result type	JPA standard
`any()` or `some()`	Logical predicate	`Boolean`	✗
`every()` or `all()`	Logical predicate	`Boolean`	✗

We may write, for example, `every(p.amount < 1000.0)`.

Below, we'll meet the ordered set aggregate functions.

Aggregate functions usually appear in the `select` clause, but control over aggregation is the responsibility of the `group by` clause, as described below.

4.3.3. Aggregate functions and collections

The `elements()` and `indices()` functions we met earlier let us apply aggregate functions to a collection:

New syntax	Legacy HQL function 💀	Applies to	Purpose
`max(elements(x))`	`maxelement(x)`	Any collection with sortable elements	The maximum element or map value
`min(elements(x))`	`minelement(x)`	Any collection with sortable elements	The minimum element or map value
`sum(elements(x))`	—	Any collection with numeric elements	The sum of the elements or map values
`avg(elements(x))`	—	Any collection with numeric elements	The average of the elements or map values
`max(indices(x))`	`maxindex(x)`	Indexed collections (lists and maps)	The maximum list index or map key
`min(indices(x))`	`minindex(x)`	Indexed collections (lists and maps)	The minimum list index or map key
`sum(indices(x))`	—	Indexed collections (lists and maps)	The sum of the list indexes or map keys
`avg(indices(x))`	—	Indexed collections (lists and maps)	The average of the list indexes or map keys

These operations are mostly useful when working with `@ElementCollection`s.

```
select title, max(indices(authors))+1, max(elements(editions)) from Book
```

4.3.4. Aggregate functions with restriction

All aggregate functions support the inclusion of a *filter clause*, a sort of mini-`where` applying a restriction to just one item of the select list:

```
select
    year(item.order.dateTime) as year,
    sum(item.quantity) filter (where not item.order.fulfilled) as unfulfilled,
    sum(item.quantity) filter (where item.order.fulfilled) as fulfilled,
    sum(item.quantity * item.book.price) filter (where item.order.paid)
from Item as item
where item.book.isbn = :isbn
group by year(item.order.dateTime)
```

The BNF for the `filter` clause is simple:

```
filterClause
    : "FILTER" "(" "WHERE" predicate ")"
```

4.3.5. Ordered set aggregate functions

An *ordered set aggregate function* is a special aggregate function which has:

- not only an optional filter clause, as above, but also

- a `within group` clause containing a mini-`order by` specification.

The BNF for `within group` is straightforward:

```
withinGroupClause
    : "WITHIN" "GROUP" "(" "ORDER" "BY" sortSpecification ("," sortSpecification)* ")"
```

There are two main types of ordered set aggregate function:

- an *inverse distribution function* calculates a value that characterizes the distribution of values within the group, for example, `percentile_cont(0.5)` is the median, and `percentile_cont(0.25)` is the lower quartile.
- a *hypothetical set function* determines the position of a "hypothetical" value within the ordered set of values.

The following ordered set aggregate functions are available on many platforms:

Type	Functions
Inverse distribution functions	`mode()`, `percentile_cont()`, `percentile_disc()`
Hypothetical set functions	`rank()`, `dense_rank()`, `percent_rank()`, `cume_dist()`
Other	`listagg()`

This query calculates the median price of a book:

```
select percentile_cont(0.5)
    within group (order by price)
from Book
```

This query finds the percentage of books with prices less than 10 dollars:

```
select 100 * percent_rank(10.0)
    within group (order by price)
from Book
```

Actually, the most widely-supported ordered set aggregate function is one which builds a string by concatenating the values within a group. This function has different names on different databases, but HQL abstracts these differences, and—following ANSI SQL—calls it `listagg()`.

```
select listagg(title, ', ')
    within group (order by isbn)
from Book
group by element(authors)
```

This very useful function produces a string by concatenation of the aggregated values of its argument.

4.3.6. Window functions

A *window function* is one which also has an `over` clause, for example:

```
select
    item.order.dateTime,
    sum(item.quantity)
        over (order by item.order.dateTime)
        as runningTotal
from Item item
```

This query returns a running total of sales over time. That is, the `sum()` is taken over a window comprising the current row of the result set, together with all previous rows.

A window function application may optionally specify any of the following clauses:

Optional clause	Keyword	Purpose
Partitioning of the result set	`partition by`	Very similar to `group by`, but doesn't collapse each partition to a single row

Optional clause	Keyword	Purpose
Ordering of the partition	`order by`	Specifies the order of rows within a partition
Windowing	`range`, `rows`, or `groups`	Defines the bounds of a window frame within a partition
Restriction	`filter`	As aggregate functions, window functions may optionally specify a filter

For example, we may partition the running total by book:

```
select
    item.book.isbn,
    item.order.dateTime,
    sum(item.quantity)
        over (partition by item.book
              order by item.order.dateTime)
        as runningTotal
from Item item
```

Every partition runs in isolation, that is, rows can't leak across partitions.

The full syntax for window function application is amazingly involved, as shown by this BNF:

```
overClause
    : "OVER" "(" partitionClause? orderByClause? frameClause? ")"

partitionClause
    : "PARTITION" "BY" expression ("," expression)*

frameClause
    : ("RANGE"|"ROWS"|"GROUPS") frameStart frameExclusion?
    | ("RANGE"|"ROWS"|"GROUPS") "BETWEEN" frameStart "AND" frameEnd frameExclusion?

frameStart
    : "CURRENT" "ROW"
    | "UNBOUNDED" "PRECEDING"
    | expression "PRECEDING"
    | expression "FOLLOWING"

frameEnd
    : "CURRENT" "ROW"
    | "UNBOUNDED" "FOLLOWING"
    | expression "PRECEDING"
    | expression "FOLLOWING"

frameExclusion
    : "EXCLUDE" "CURRENT" "ROW"
    | "EXCLUDE" "GROUP"
    | "EXCLUDE" "TIES"
    | "EXCLUDE" "NO" "OTHERS"
```

Window functions are similar to aggregate functions in the sense that they compute some value based on a "frame" comprising multiple rows. But unlike aggregate functions, window functions don't flatten rows within a window frame.

Window frames

The *window frame* is the set of rows within a given partition that is passed to the window function. There's a different window frame for each row of the result set. In our example, the window frame comprised all the preceding rows within the partition, that is, all the rows with the same `item.book` and with an earlier `item.order.dateTime`.

The boundary of the window frame is controlled via the windowing clause, which may specify one of the following modes:

Mode	Definition	Example	Interpretation
`rows`	Frame bounds defined by a given number of rows	`rows 5 preceding`	The previous 5 rows in the partition
`groups`	Frame bounds defined by a given number of *peer groups*, rows belonging to the same peer group if they are assigned the same position by `order by`	`groups 5 preceding`	The rows in the previous 5 peer groups in the partition

Mode	Definition	Example	Interpretation
`range`	Frame bounds defined by a maximum difference in *value* of the expression used to `order by`	`range between 1.0 preceding and 1.0 following`	The rows whose `order by` expression differs by a maximum absolute value of `1.0` from the current row

The frame exclusion clause allows excluding rows around the current row:

Option	Interpretation
`exclude current row`	Excludes the current row
`exclude group`	Excludes rows of the peer group of the current row
`exclude ties`	Excludes rows of the peer group of the current row except the current row
`exclude no others`	The default, does not exclude anything

By default, the window frame is defined as `rows between unbounded preceding and current row exclude no others`, meaning every row up to and including the current row.

The modes `range` and `groups`, along with frame exclusion modes, are not available on every database.

Widely supported window functions

The following window functions are available on all major platforms:

Window function	Purpose	Signature
`row_number()`	The position of the current row within its frame	`row_number()`
`lead()`	The value of a subsequent row in the frame	`lead(x)`, `lead(x, i, x)`
`lag()`	The value of a previous row in the frame	`lag(x)`, `lag(x, i, x)`
`first_value()`	The value of a first row in the frame	`first_value(x)`
`last_value()`	The value of a last row in the frame	`last_value(x)`
`nth_value()`	The value of the `n`th row in the frame	`nth_value(x, n)`

In principle every aggregate or ordered set aggregate function might also be used as a window function, just by specifying `over`, but not every function is supported on every database.

Window functions and ordered set aggregate functions aren't available on every database. Even where they are available, support for particular features varies widely between databases. Therefore, we won't waste time going into further detail here. For more information about the syntax and semantics of these functions, consult the documentation for your dialect of SQL.

4.4. Operations on result sets

These operators apply not to expressions, but to entire result sets:

- `union` and `union all`,
- `intersect` and `intersect all`, and
- `except` and `except all`.

Just like in SQL, `all` suppresses the elimination of duplicate results.

```
select nomDePlume from Author where nomDePlume is not null
union
select name from Person
```

4.5. Sorting

By default, the results of the query are returned in an arbitrary order.

> Imposing an order on a set is called *sorting*.
>
> A relation (a database table) is a set, and therefore certain particularly dogmatic purists have argued that sorting has no place in the algebra of relations. We think this is more than a bit silly: practical data analysis almost always involves sorting, which is a perfectly well-defined operation.

The `order by` clause specifies a list of projected items used to sort the results. Each sorted item may be:

- an attribute of an entity or embeddable class,
- a more complex expression,
- the alias of a projected item declared in the select list, or
- a literal integer indicating the ordinal position of a projected item in the select list.

Of course, in principle, only certain types may be sorted: numeric types, string, and date and time types. But HQL is very permissive here and will allow an expression of almost any type to occur in a sort list. Even the identification variable of an entity with a sortable identifier type may occur as a sorted item.

> The JPQL specification requires that every sorted item in the `order by` clause also occur in the `select` clause. HQL does not enforce this restriction, but applications desiring database portability should be aware that some databases *do*.
>
> Therefore, you might wish to avoid the use of complex expressions in the sort list.

The BNF for a sorted item is:

```
sortExpression sortDirection? nullsPrecedence?

sortExpression
    : identifier | INTEGER_LITERAL | expression

sortDirection
    : "ASC" | "DESC"

nullsPrecedence
    : "NULLS" ("FIRST" | "LAST")
```

Each sorted item listed in the `order by` clause may explicitly specify a direction, either:

- `asc` for ascending order, or
- `desc` for descending order.

If no direction is explicitly specified, the results are returned in ascending order.

Of course, there's an ambiguity with respect to null values. Therefore, the sorting of null values may be explicitly specified:

Precedence	Interpretation
`nulls first`	Puts null values at the beginning of the result set
`nulls last`	Puts them at the end

```
select title, publisher.name
from Book
order by title, publisher.name nulls last
```

```
select book.isbn,
    year(order.dateTime) as year,
    sum(quantity) as yearlyTotalSold,
    sum(quantity * book.price) as yearlyTotalBilled
from Item
where book.isbn = :isbn
group by year(order.dateTime)
having year(order.dateTime) > 2000
```

```
   and sum(quantity) > 0
order by yearlyTotalSold desc, year desc
```

Queries with an ordered result list may have limits or pagination.

4.5.1. Limits and offsets

It's often useful to place a hard upper limit on the number of results that may be returned by a query. The `limit` and `offset` clauses are an alternative to the use of `setMaxResults()` and `setFirstResult()` respectively, and may similarly be used for pagination.

If the `limit` or `offset` is parameterized, it's much easier to use `setMaxResults()` or `setFirstResult()`.

The SQL `fetch` syntax is supported as an alternative:

Short form	Verbose form	Purpose
`limit 10`	`fetch first 10 rows only`	Limit result set
`limit 10 offset 20`	`offset 20 rows fetch next 10 rows only`	Paginate result set

The BNF gets a bit complicated:

```
limitClause
    : "LIMIT" parameterOrIntegerLiteral

offsetClause
    : "OFFSET" parameterOrIntegerLiteral ("ROW" | "ROWS")?

fetchClause
    : "FETCH" ("FIRST" | "NEXT")
      (parameterOrIntegerLiteral | parameterOrNumberLiteral "%")
      ("ROW" | "ROWS")
      ("ONLY" | "WITH" "TIES")
```

These two queries are identical:

```
select title from Book
order by title, published desc
limit 50
```

```
select title from Book
order by title, published desc
fetch first 50 rows only
```

These are well-defined limits: the number of results returned by the database will be limited to 50, as promised. But not every query is quite so well-behaved.

Limiting certainly *isn't* a well-defined relational operation, and must be used with care.

In particular, limits don't play well with fetch joins.

This next query is accepted by HQL, and no more than 50 results are returned by `getResultList()`, just as expected:

```
select title from Book
    join fetch authors
order by title, published desc
limit 50
```

However, if you log the SQL executed by Hibernate, you'll notice something wrong:

```
select
    b1_0.isbn,
    a1_0.books_isbn,
    a1_0.authors_ORDER,
    a1_1.id,
    a1_1.bio,
```

```
    a1_1.name,
    a1_1.person_id,
    b1_0.price,
    b1_0.published,
    b1_0.publisher_id,
    b1_0.title
from
    Book b1_0
join
    (Book_Author a1_0
        join
            Author a1_1
                on a1_1.id=a1_0.authors_id)
        on b1_0.isbn=a1_0.books_isbn
order by
    b1_0.title,
    b1_0.published desc
```

What happened to the `limit` clause?

When limits or pagination are combined with a fetch join, Hibernate must retrieve all matching results from the database and *apply the limit in memory*!

This *almost certainly* isn't the behavior you were hoping for, and in general will exhibit *terrible* performance characteristics.

4.6. Common table expressions

A *common table expression* or CTE may be thought of as a sort of named subquery. Any query with an uncorrelated subquery can in principle be rewritten so that the subquery occurs in the `with` clause.

But CTEs have capabilities that subqueries don't have. The `with` clause lets us:

- specify materialization hints, and
- write recursive queries.

On databases which don't support CTEs natively, Hibernate attempts to rewrite any HQL query with CTEs as a SQL query with subqueries. This is impossible for recursive queries, unfortunately.

Let's take a quick look at the BNF:

```
withClause
    : "WITH" cte ("," cte)*

cte
    : identifier AS ("NOT"? "MATERIALIZED")? "(" queryExpression ")"
      searchClause? cycleClause?
```

The `with` clause comes right at the start of a query. It may declare multiple CTEs with different names.

```
with
    paid as (
        select ord.id as oid, sum(payment.amount) as amountPaid
        from Order as ord
            left join ord.payments as payment
        group by ord
        having local datetime - ord.dateTime < 365 day
    ),
    owed as (
        select ord.id as oid, sum(item.quantity*item.book.price) as amountOwed
        from Order as ord
            left join ord.items as item
        group by ord
        having local datetime - ord.dateTime < 365 day
    )
select id, paid.amountPaid, owed.amountOwed
from Order
where paid.amountPaid < owed.amountOwed
  and paid.oid = id and owed.oid = id
```

Notice that if we rewrote this query using subqueries, it would look quite a lot clumsier.

4.6.1. Materialization hints

The `materialized` keyword is a hint to the database that the subquery should be separately executed and its results stored in a temporary table.

On the other hand, its nemesis, `not materialized`, is a hint that the subquery should be inlined at each use site, with each usage optimized independently.

> The precise impact of materialization hints is quite platform-dependant.

Our example query from above hardly changes. We just add `materialized` to the CTE declarations.

```
with
    paid as materialized (
        select ord.id as oid, sum(payment.amount) as amountPaid
        from Order as ord
            left join ord.payments as payment
        group by ord
        having local datetime - ord.dateTime < 365 day
    ),
    owed as materialized (
        select ord.id as oid, sum(item.quantity*item.book.price) as amountOwed
        from Order as ord
            left join ord.items as item
        group by ord
        having local datetime - ord.dateTime < 365 day
    )
select id, paid.amountPaid, owed.amountOwed
from Order
where paid.amountPaid < owed.amountOwed
  and paid.oid = id and owed.oid = id
```

4.6.2. Recursive queries

A *recursive query* is one where the CTE is defined self-referentially. Recursive queries follow a very particular pattern. The CTE is defined as a union of:

- a base subquery returning an initial set of rows where the recursion begins,
- a recursively-executed subquery which returns additional rows by joining against the CTE itself.

Let's demonstrate this with an example.

First we'll need some sort of tree-like entity:

```
@Entity
class Node {
    @Id Long id;
    String text;
    @ManyToOne Node parent;
}
```

We may obtain a tree of `Node`s with the following recursive query:

```
with Tree as (
    /* base query */
    select root.id as id, root.text as text, 0 as level
        from Node root
        where root.parent is null
    union all
    /* recursion */
    select child.id as id, child.text as text, level+1 as level
        from Tree parent
        join Node child on child.parent.id = parent.id
)
select text, level
from Tree
```

When querying a tree-like of data structure, the base subquery usually returns the root node or nodes. The recursively-executed subquery returns the children of the current set of nodes. It's executed repeatedly with the results of the previous execution. Recursion terminates when the recursively-executed subquery returns no new nodes.

Hibernate cannot emulate recursive queries on databases which don't support them natively.

Now, if a graph contains cycles, that is, if it isn't a tree, the recursion might never terminate.

4.6.3. Cycle detection

The `cycle` clause enables cycle detection, and aborts the recursion if a node is encountered twice.

```
with Tree as (
    /* base query */
    select root.id as id, root.text as text, 0 as level
        from Node root
        where root.parent is null
    union all
    /* recursion */
    select child.id as id, child.text as text, level+1 as level
        from Tree parent
        join Node child on child.parent.id = parent.id
) cycle id set abort to 'aborted!' default ''  /* cycle detection */
select text, level, abort
from Tree
order by level
```

Here:

- the `id` column is used to detect cycles, and
- the `abort` column is set to the string value `'aborted!'` if a cycle is detected.

Hibernate emulates the `cycle` clause on databases which don't support it natively.

The BNF for `cycle` is:

```
cycleClause
    : "CYCLE" identifier ("," identifier)*
      "SET" identifier ("TO" literal "DEFAULT" literal)?
      ("USING" identifier)?
```

The column optionally specified by `using` holds the path to the current row.

4.6.4. Ordering depth-first or breadth-first

The `search` clause allows us to control whether we would like the results of our query returned in an order that emulates a depth-first recursive search, or a breadth-first recursive search.

In our query above, we explicitly coded a `level` column that holds the recursion depth, and ordered our result set according to this depth. With the `search` clause, that bookkeeping is already taken care of for us.

For depth-first search, we have:

```
with Tree as (
    /* base query */
    select root.id as id, root.text as text
        from Node root
        where root.parent is null
    union all
    /* recursion */
    select child.id as id, child.text as text
        from Tree parent
        join Node child on child.parent.id = parent.id
) search depth first by id set level  /* depth-first search */
from Tree
select text
order by level
```

And for breadth-first search, we only need to change a single keyword:

```
with Tree as (
    /* base query */
    select root.id as id, root.text as text
        from Node root
        where root.parent is null
    union all
    /* recursion */
    select child.id as id, child.text as text
        from Tree parent
        join Node child on child.parent.id = parent.id
) search breadth first by id set level  /* breadth-first search */
from Tree
select text
order by level desc
```

Hibernate emulates the `search` clause on databases which don't support it natively.

The BNF for `search` is:

```
searchClause
    : "SEARCH" ("BREADTH"|"DEPTH") "FIRST"
      "BY" searchSpecifications
      "SET" identifier

searchSpecifications
    : searchSpecification ("," searchSpecification)*

searchSpecification
    : identifier sortDirection? nullsPrecedence?
```

Chapter 5. Credits

The full list of contributors to Hibernate ORM can be found on the GitHub repository.

The following contributors were involved in this documentation:

- Gavin King