JAKARTA EE

Jakarta Query

1.0, September 07, 2025: Draft

Table of Contents

Specification: Jakarta Query

Version: 1.0

Status: Draft

Release: September 07, 2025

Copyright

Eclipse Foundation Specification License - v1.1

Disclaimers

Chapter 1. Introduction

Jakarta Query defines an object-oriented query language designed for use with Jakarta Persistence, Jakarta Data, and Jakarta NoSQL, with:

- a core language which can be implemented by Jakarta Data and Jakarta NoSQL providers, and
- an extended language tailored for Jakarta Persistence providers or other persistence technologies backed by relational databases.

The language is closely based on the existing query languages defined by Jakarta Persistence and Jakarta Data, and is backward compatible with both.

Jakarta Query prioritizes clients written in Java. However, it is not by nature limited to Java, and implementations in other sufficiently Java-like programming languages are encouraged.

1.1. Object-oriented query languages

A data structure in an object-oriented language is a graph of objects interconnected by unidirectional object references, which may be polymorphic. Some non-relational databases support similar representations. On the other hand, relational databases represent relationships between entities using foreign keys, and therefore SQL has no syntactic construct representing navigation of an association. Similarly, inheritance and polymorphism can be easily represented within the relational model, but are not present as first-class constructs in the SQL language. An object-oriented query language is a dialect of SQL with support for associations and subtype polymorphism.

1.2. Historical background

Object-oriented dialects of SQL have existed since at least the early 1990s. The Object Query Language (OQL) was an early example, targeting object databases, but was never widely used, since object databases were themselves not widely adopted. Hibernate Query Language (HQL) and the Enterprise JavaBeans Query Language (EJB-QL) were both introduced in 2001 as query languages intended for use with object/relational mapping. HQL was widely adopted by the Java community and was eventually standardized as the Java Persistence Query Language (JPQL) by JSR-220 in 2006. JPQL has been implemented by at least five different products and is in extremely wide use today. On the other hand, since JPQL is defined as part of the Jakarta Persistence specification, it has not been reused outside the context of object/relational mapping in Java. In 2024, Jakarta Data 1.0 introduced the Jakarta Data Query Language (JDQL), a strict subset of JPQL intended for use with non-relational databases.

It is now inconvenient that JDQL and JPQL are maintained separately by different groups, and so the Jakarta Query project has taken on responsibility for their evolution.

1.3. Goals

This specification defines an object-oriented query language with two well-defined levels of compliance:

- the *extended language*, which is known to be implementable by persistence solutions backed by SQL databases, and
- the *core language*, a strict subset which is designed to be implementable on other kinds of non-SQL datastores.

The extended language is designed for reuse by Jakarta Persistence. The core language is designed for reuse by Jakarta Data and Jakarta NoSQL. Jakarta Query itself has no dependence on either of these specifications, and reuse in other contexts is encouraged.

This document:

- standardizes the syntax and semantics of the language,
- delineates the core subset,
- provides guidelines on how query language constructs map to language elements in a program written in Java.

The definition of the query language itself is independent of the Java programming language, and of any details of the underlying datastore and data access technology.

1.4. Non-Goals

This specification does not specify Java APIs for:

- executing queries,
- embedding queries in Java programs,
- constructing queries programmatically, nor
- defining entity classes which are used in queries.

Jakarta Persistence and Jakarta Data define diverse ways in which queries may be embedded and executed in Java, using the `EntityManager` or a `@Repository` interface, respectively.

Furthermore, the semantics defined by Jakarta Query may be reused by reference in other specifications, for example, in the definition of the Jakarta Persistence Criteria API, or in the definition of the Jakarta Data Restrictions.

This document does not define how constricts in Jakarta Query map to constructs in SQL or in any other datastore-specific query languages. Jakarta Persistence defines an interpretation compatible with SQL.

1.5. Conventions

ANTLR 4-style BNF is used to define the syntax of the language.

1.6. Jakarta Query Project Team

This specification is being developed as part of Jakarta Query project under the Jakarta EE Specification Process. It is the result of the collaborative work of the project committers and various contributors.

1.6.1. Project Leads

- Gavin King
- Lukas Jungmann

1.6.2. Committers

- Gavin King
- Lukas Jungmann
- Nathan Rauh
- Riva Tholoor Philip

1.6.3. Contributors

The complete list of Jakarta Query contributors may be found here.

Chapter 2. Type system

This specification assumes that data is represented in a structured form, that is, that it is expressed in terms of:

1. **atomic values**,
2. **collections**, and
3. **structures**.

Any atomic value, collection, or structure belongs to at least one well-defined type.

2.1. Atomic values

Atomic values are things defined externally to this specification, taken to include—at bare minimum—strings of Unicode characters, `true` and `false`, integers, floating point numbers, and dates, times, and datetimes. Every atomic value has a named type, referred to in the sister-specifications of this specification as a *basic type*.

The atomic value types allowed by an implementation of Jakarta Query depend on the programming language and underlying database technology.

Atomic values of distinct type are never considered identical (equal). Whether two atomic values of the same type are considered identical depends on semantics which are specific to the type. For example, two Unicode strings are identical if they have the same length and the same Unicode character at each position. Identity for a given atomic value type must be reflexive, symmetric, and transitive.

2.2. Collections

A *collection* is a finite set, list, or map containing atomic values or structures. A collection might contain atomic values, or it might contain structures. The keys of a map must be atomic values.

Collections are homogeneous. A collection may not contain both atomic values and structures.

- If a collection contains atomic values, then every atomic value has the same type.
- If a collection contains structures, then every structure has the same type.
- As an exception to the previous rule, a collection may contain structures belonging to distinct entity types if there is some entity type which is inherited by the type of every structure contained in the collection.
- Every key of a map has the same atomic value type.

A collection therefore has a type derived from the type of the atomic values or structures it contains—and, for a map, from the type of its keys. We consider that two collections have the same type if they contain the same type of atomic value or structure. For a map, we also require that they have the same type of key.

A collection has no identity independent of the values it contains.

- Two sets are identical if they contain exactly the same elements.
- Two lists are identical if they contain exactly the same elements at exactly the same positions.
- Two maps are identical if they contain exactly the same keys, and identical values for each key.

2.3. Structures and records

A *structure* is a finite set of labelled elements, each of which is an atomic value, a nested structure, or a collection.

Structures are inhomogeneous. Elements with distinct labels may have distinct types.

Every structure has a type derived from the types of its elements. We consider that two structures have the same type if:

- they contain the same labels, and
- for each label, the labelled elements have the same type.

A structure has no identity independent of the labelled elements it contains. Two structures are identical if they contain exactly the same labels, and identical atomic values, collections, or nested structures for each label.

For example—borrowing a convenient notation—the following structure might represent a book:

```
{
  isbn: "1-85723-235-6",
  title: "Feersum Endjinn",
  pages: 279,
  year: 1994,
  authors: [
    {
      name: "Iain M. Banks",
      born: 'February 16, 1954',
      died: 'June 9, 2013'
    }
  ]
}
```

A *record* is a special kind of structure. Every record must have an *identifier* which uniquely distinguishes it from other structures with the same type. The identifier must be a subset of the labelled elements of the structure.

For example, `{ isbn: "1-85723-235-6" }` might be an identifier of the structure given above.

2.4. Entities and embeddable types

A structure type might be assigned a static name.[1]

- An *embeddable type* is an assignment of a name to a given type of structure. For example, we might assign the name `Author` to the type of the structure:

```
{
  name: "Iain M. Banks",
  born: 'February 16, 1954',
  died: 'June 9, 2013'
}
```

- An *entity* (or *entity type*) is an assignment of a name to a given type of record. For example, we might assign the name `Book` to the type of the record given in the example above.

The name must be a legal Java identifier. That is, it must be a string of letters and digits, along with the characters _ and $, and must begin with a letter or with _.[2]

It follows from this definition that we may express:

- an embeddable type as a set of labels, together with their types,
- an entity as a set of labels, together with their types, and whether they belong to the record identifier.

For example:

```
Author :=
{
    name: String,
```

```
    born: LocalDate,
    died: LocalDate
}

Book :=
{
  isbn: @Id String,
  title: String,
  pages: Integer,
  year: Integer,
  authors: Set<Author>
}
```

A record with the same type as an entity type is said to be an *instance* (or *instantiation*) of the entity. Similarly, a structure with the same type as an embeddable type is said to be an instance of the embeddable type. [3]

An entity is directly addressable in a query. An atomic value type, a collection type, or a structure type which is not an entity is not directly addressable, and must be addressed indirectly via an entity.

Some database technologies are capable of storing an arbitrary structure whose type is not known at compile time. Other technologies require that the structure belong to a defined entity or embeddable type. Independent of the database technology itself, an implementation of Jakarta Query might require that structure types be named, or might offer a way to encode and store generic structures. Implementations of Jakarta Query are not required to support storage of such generic structures.

The name of an entity might be involved in mapping an association between a type defined in a programming language (for example, a Java class) and an area of storage in the database (for example, a table). Such mappings are completely outside the scope of this specification.

2.4.1. Entity type inheritance

Inheritance is a relationship between entity types. An entity `X` inherits an entity `Y` if and only if for every type labelled `y` in `Y`, there is a corresponding type labelled `y` in `X` and either:

- the two types with label `y` are identical, or
- the type labelled `y` in `Y` is an entity type `T`, the type labelled `y` in `X` is an entity type `S`, and `S` inherits `T`.

Thus, there is a simple mapping from records of type `X` to records of type `Y`. Given a record *r* of type `X`, the *restriction* of *r* to a type `Y` inherited by `X` is a structure *s* containing an element labelled `y` for each type with label `y` occurring in the type `Y`:

- If the type of the element *e* of `X` with label `y` is identical to the type with label `y` in `Y`, then *s* contains *e* labelled `y`.
- Otherwise, the type of the element *e* of `X` must be an entity type `S`, the type with the label `y` in `Y` must be an entity type `T`, and `S` must inherit `T`. Then *s* contains the restriction of *e* to `T`, labelled `y`.

Then *s* is a record of type `Y`.

Any well-defined operation on records of type `Y` is also a well-defined operation on the restriction of a record to `Y`. We therefore adopt the principle that an operation which may be applied to a record of a given entity type may also be applied to a record of any entity type which inherits the first entity type.

2.5. Circularity

Our definitions above are intended to be descriptive rather than constructive. It's not, in general, possible to construct

an arbitrary record in a finite number of steps by beginning with atomic values and then recursively constructing structures and collections.

The reason for this is that the graph representing a record is not, in general, a finite tree. The representation of a record as a tree might necessarily be infinite, with a nonterminating cycle involving two or more structures.

On the other hand, any record is assumed to be representable as a finite directed graph.

2.6. Databases

A *database* is a finite set of records.

A given database might be restricted to contain only records belonging to a statically-enumerated list of entities.

Some databases store records as trees; other databases store them in a flattened *normalized* form. In some databases, records must be disjoint; in other databases, one record might be nested inside another record. Questions about the representation used for record storage are completely outside the scope of this specification. Such questions are the domain of our sister-specifications.

[1] That is, the name is assigned to the type before the program using Jakarta Query is compiled and executed.

[2] Use of `_` or `$` in the name of an entity is discouraged.

[3] In some implementations, it might be possible to assign multiple names to a single structure type, and then a given instance of that type might be considered to belong to just one of the named entity or embeddable types. We do not address this wrinkle here, since implementations of Jakarta Query are not required to allow this.

Chapter 3. Basic operations

A query is a sequence of operations acting on lists of structures:

- specification of an initial **root entity**,
- a sequence of **joins** to joined entities, collections, or embeddable types,
- **restriction**,
- **aggregation**,
- more **restriction**,
- **projection**, and
- **ordering**.

For queries without nesting, these operations may be viewed as a pipeline.[1] Each operation may then be thought of as a stage of the pipeline. Some stages may be missing in a given query. The only required stage is the first stage: specification of the root entity.

Each operation in the pipeline produces a *result list*. A result list is a list in the sense of the previous chapter. The elements of a result list are always structures. Thus, every result list has a well-defined type, according to the rules specified above.

Each operation from the list above—except for the first operation, specification of the root entity—acts on the output of the previous stage. When evaluated, the operation produces its result list by transforming the result list produced by the previous stage.

A join or root entity has access not only to the result list of its previous stage, but also to the content of a database. A query is executed against a specific database and any operation which ranges over an entity type ranges only over those records which belong to the database. From now on, the database itself will usually be implicit, and we will not explicitly specify that the records under consideration must belong to the database.

3.1. The root entity

Every query begins with a *root entity*. A root entity is specified using the syntax:

```
from X
```

where `X` is the name of the entity.

We may assign a label to the root entity of the query using the syntax:

```
from X as x
```

which may be abbreviated as:

```
from X x
```

The label `x` must be a legal Java identifier. This label is often called an *alias* or *identification variable*.

When no label is explicitly specified, the root entity is assigned the implicit label `this`.

A root entity specification evaluates to a result list containing a structure for each record of the root entity type or of any entity type which inherits the root entity type. Each structure contains a single labelled element: the record, labelled by the alias `x`.

That is, for each record *r* in the database, the result list has a structure *s* containing *r* labelled by `x` if and only if the type

of *r* inherits the root entity type. The structure *s* contains no other elements.

For example, the query `from Book` might return a list containing structures like:

```
{
  this: {
    isbn: "9781857232738",
    title: "Feersum Endjinn",
    pages: 279,
    year: 1994,
    authors: [
      {
        name: "Iain M. Banks",
        born: 'February 16, 1954',
        died: 'June 9, 2013'
      }
    ]
  }
}
```

And the query `from Book as book` returns a list containing structures like:

```
{
  book: {
    isbn: "9781857232738",
    title: "Feersum Endjinn",
    pages: 279,
    year: 1994,
    authors: [
      {
        name: "Iain M. Banks",
        born: 'February 16, 1954',
        died: 'June 9, 2013'
      }
    ]
  }
}
```

3.2. Joins

Every join has a *target*, which must be:

- an entity,
- a collection, or
- an embeddable type.

A join introduces a new labelled element to a result list. Evaluating a sequence of joins produces a result list in which each structure contains a labelled element corresponding to each join. The label of this element is often called the *alias* or *identification variable* of the join.

3.2.1. Joins to named entities

A join may specify the name of an entity:

```
[from], Y as y
```

where `Y` is the name of the entity, `y` is the label (alias) assigned to it, and `[from]` is a root entity or a join. As before, the keyword `as` is optional.

This kind of join produces a cartesian product. For each structure *r* of the result list of the operation on which the join acts, and for each record *y* of any entity type which inherits `Y`, the result list contains a structure *r′* containing all labelled elements of *r* together with the record *y* labelled by the alias `y`.

For example, this query is a cartesian product:

```
from Book as book, Loan as loan
```

This query evaluates to a result list containing a structure for each pairing of a record of entity `Book` with a record of entity `Loan`. The structure contains two labelled elements, one for each entity, each labelled by its corresponding alias, `book`, and `loan`, respectively. The list might contain structures this:

```
{
  book: {
    isbn: "9781857232738",
    title: "Feersum Endjinn",
    pages: 279,
    year: 1994,
    authors: [
      {
        name: "Iain M. Banks",
        born: 'February 16, 1954',
        died: 'June 9, 2013'
      }
    ]
  },
  loan: {
    bookIsbn: "9781932394153",
    borrowerCard: "XYZ-123"
  }
}
```

Note that there is no meaningful relationship between the `book` and the `loan`.

A join to a named entity may be immediately followed by a restriction. In this case, the syntax is slightly different:

```
[from] join Y as y on [predicate]
```

where `[predicate]` is a predicate, as defined later in [predicates].

For example:

```
from Book as book
join Loan as loan
    on book.isbn = loan.bookIsbn
```

This kind of join is interpreted as a sequence of two operations, a join of the previous kind, with no `on`, followed by a restriction with the given predicate.

The result of the query might contain structures like:

```
{
  book: {
    isbn: "9781857232738",
    title: "Feersum Endjinn",
    pages: 279,
    year: 1994,
    authors: [
      {
        name: "Iain M. Banks",
```

```
        born: 'February 16, 1954',
        died: 'June 9, 2013'
      }
    ]
  },
  loan: {
    bookIsbn: "9781857232738",
    borrowerCard: "ABC-098"
  }
}
```

This time, `isbn` and `bookIsbn` agree.

3.2.2. Joins to nested entities, collections, or collections

Instead of named entity, a join may identify a structure or collection nested within the result list of the operation on which it acts:

```
[from] inner join [path] as y
```

where `[path]` is a path expression, as defined later in [path-expressions], and `y` is the label.

As usual, the keyword `as` is optional. The keyword `inner` is also completely optional, and so a join may be written:

```
[from] join [path] as y
```

For example:

```
from Book as book
join book.authors as author
```

The path expression identifies a structure nested within the result list of the operation on which the join acts.

For each structure *r* of the result list of the operation on which the join acts:

- If the path expression resolves to a structure *s*, the result list contains a structure *r′* containing all labelled elements of *r* together with the structure *s* labelled by the alias `y`.
- If the path expression resolves to a collection *c*, the result list contains, for each element *e* of *c*, a structure *r′* containing all labelled elements of *r* together with the structure *e* labelled by the alias `y`.

The previous example evaluates to a list containing a structure for each `Author` of each `Book`. The structure contains two labelled elements, one for each entity, each labelled by its corresponding alias, `book`, and `author`, respectively. The list might contain structures like this:

```
{
  book: {
    isbn: "9781857232738",
    title: "Feersum Endjinn",
    pages: 279,
    year: 1994,
    authors: [
      {
        name: "Iain M. Banks",
        born: 'February 16, 1954',
        died: 'June 9, 2013'
      }
    ]
  },
  author: {
```

```
    name: "Iain M. Banks",
    born: 'February 16, 1954',
    died: 'June 9, 2013'
  }
}
```

Notice that this kind of join has the effect of duplicating nested structures or atomic values at the top level of the of structure belonging to the result list.

This picture should not be taken too literally. Implementations of Jakarta Query do not, in practice, always return the entire result of a query to the client, but instead replace some branches of the graph with some sort of proxy object.

3.2.3. Left joins

A left join is similar to a regular join:

```
[from] left outer join [path] as y
```

where `[path]` is a path expression, as before, and `y` is the label.

As usual, the keyword `as` is optional. The keyword `outer` is also completely optional, and so a left join may be written:

```
[from] left join [path] as y
```

For example:

```
from Book as book
left join book.authors as author
```

The path expression identifies a structure nested within the result list of the operation on which the join acts.

For each structure *r* of the result list of the operation on which the join acts:

- If the path expression resolves to a structure *s*, the result list contains a structure *r′* containing all labelled elements of *r* together with the structure *s* labelled by the alias `y`.
- If the path expression resolves to a nonempty collection *c*, the result list contains, for each element *e* of *c*, a structure *r′* containing all labelled elements of *r* together with the structure *e* labelled by the alias `y`.
- Otherwise, if a path expression resolves to no structure, or to an empty collection, the result list contains a structure *r′* containing only the labelled elements of *r*.

3.3. Restriction

Restriction, also called *selection*, reduces the size of a result list, without modifying its type.

Restriction may occur before or after aggregation, or, as we already saw above, it may occur immediately after a join.

When restriction precedes aggregation, the syntax is:

```
[from] where [predicate]
```

where `[predicate]` is a logical predicate expression.

When restriction follows aggregation, the syntax is:

```
[group-by] having [predicate]
```

where `[group-by]` is a legal aggregation.

Restriction eliminates every element of the result list which does not satisfy the given predicate expression, as defined later in [predicates]. That is, the result list of a restriction contains a structure *r* if and only if:

- *r* is in the result list of the operation on which the restriction acts, and
- *r* satisfies the logical predicate.

3.3.1. Restriction and aggregation

When restriction is applied to a query involving aggregation, the predicate may only involve:

- value expressions which also occur in the `group by` clause, and
- aggregate function expressions, as specified below in [aggregate-functions].

In this case, the restriction eliminates entire nested lists belonging to the result list of the aggregation operation.

3.4. Aggregation

Aggregation groups the elements of a result list into sublists. That is, it transforms a list into a list of lists.

Aggregation follows a root entity or join:

```
[from] group by [expression], [expression], ...
```

where each `[expression]` is a value expression, as defined later in [expressions].

1. For each structure *r* of the result list of the operation on which the aggregation acts, a *grouping tuple* is constructed by evaluating each of the value expressions specified by the aggregation in the context of the structure *r*, and packaging the resulting atomic values in a structure *t* where each value is labeled by the position of the value expression in the `group by` clause.
2. For each distinct resulting value *t* of the grouping tuple, a nested list l_t is constructed containing every structure *r* which produced that value of the grouping tuple.
3. Finally, the result list of the aggregation contains every such nested list l_t.

Each value expression must evaluate to an atomic value or record.

If a value expression evaluates to a record, the record may be replaced by its identifier in the grouping tuple.

3.5. Projection

Projection changes the type of a result list without modifying its size.

A projection is written in the form:

```
[result] select [expression] as x, [expression] as y, ...
```

or, more conventionally, but much more confusingly, in the form:

```
select [expression] as x, [expression] as y, ... [result]
```

where `x`, `y`, ... are all labels and `[result]` is a root entity, join, restriction, or aggregation, and each `[expression]` is a value expression, as defined later in [expressions].

As usual, the `as` keyword is optional, and the labels must be legal Java identifiers.

The labels, sometimes called *aliases*, are optional. If a label is missing from a value expression, the value expression is automatically assigned a label.

For historical reasons, the label defaults to the integer position of the value expression in the `select` list. This is unfortunate because an integer is not a legal Java identifier, and therefore not a legal label. Such defaulted labels may not be referred to in the query language except—again for historical reasons—in the `order by` clause.

Projection produces a new structure r' for each structure r in the result list of the operation on which the projection acts. The new structure r' is built by evaluating the value expressions specified by the projection in the context of the corresponding element structure r, according to semantics given later in [expressions]. For each value expression with label `x` in the given `select` list, r' contains a element labelled `x` obtained by evaluating the value expression in the context of r.

For example:

```
from Book as book
join book.authors as author
select book.isbn as isbn, book.title as title, author.name as author
```

returns a list containing elements like:

```
{
    isbn: "9781857232738",
    title: "Feersum Endjinn",
    author: "Iain M. Banks"
}
```

3.5.1. Projection and aggregation

When projection is applied to a query involving aggregation, every value expression in the `select` list must be either:

- a value expression which also occurs in the `group by` clause, or
- an aggregate function expression, as specified below in [aggregate-functions].

In this case, the projection has the additional effect of collapsing the list of lists produced by aggregation, producing a single result structure for each nested list in the result list of the operation to which the projection applies.

3.6. Ordering

Ordering changes the order of the elements in a result list, without changing the size or type of the list.

Ordering is the last operation of a query:

```
[result] order by [order], [order], ...
```

where `[result]` is a root entity, join, restriction, aggregation, or projection, and each `[order]` is an ordering criterion comprising:

- a value expression, subject to the restrictions given below, and

- optionally, `asc` or `desc`, specifying ascending or descending order, and
- optionally, `nulls first`, or `nulls last`, specifying the precedence of null values.

If neither `asc` nor `desc` is explicitly specified, ascending order is assumed. If neither `nulls first` nor `nulls last` is explicitly specified, the precedence of null values is not defined by this specification.

1. For each structure *r* of the result list of the operation on which the aggregation acts, an *ordering tuple* is constructed by evaluating each of the value expressions specified by the ordering operation in the context of the structure *r*, and packaging the resulting atomic values in a structure *t* where each value is labeled by the position of the value expression in the `group by` clause.
2. The result list is sorted according to the lexicographic order of the resulting ordering tuples.

This specification does not specify an order for atomic values or structures. Such ordering is typically determined by the database itself.

Each value expression in the `order by` list must also occur in the projection list. TODO!

For example:

```
from Book as book
join book.authors as author
select book.isbn as isbn, book.title as title, author.name as author
order by book.isbn desc
```

[1] Subqueries complicate the picture; a query involving subqueries is conceptually a tree.

Chapter 4. Jakarta Query Language

The Jakarta Query Language (JQL) is a simple language designed to be used inside the `@Query` annotation to specify the semantics of query methods of Jakarta Query repositories. The language is in essence a subset of the widely-used Jakarta Persistence Query Language (JPQL), and thus a dialect of SQL. But, consistent with the goals of Jakarta Query, it is sufficiently limited in functionality that it is easily implementable across a wide variety of data storage technologies. Thus, the language defined in this chapter excludes features of JPQL which, while useful when the target datasource is a relational database, cannot be easily implemented on all non-relational datastores. In particular, the `from` clause of a Jakarta Query query may contain only a single entity.

A Jakarta Query provider backed by access to a relational database might choose to allow the use of a much larger subset of JPQL—or even the whole language—via the `@Query` annotation. Such extensions are not required by this specification.

4.1. Type system

Every expression in a JQL query is assigned a Java type. An implementation of JQL is required to support the Java types listed in [Basic types], that is: primitive types, `String`, `LocalDate`, `LocalDateTime`, `LocalTime`, `Year`, and `Instant`, `java.util.UUID`, `java.math.BigInteger` and `java.math.BigDecimal`, `byte[]`, and `enum` types.

An implementation of JQL is permitted and encouraged to support additional types. Use of such types is not guaranteed to be portable between implementations.

The interpretation of an operator expression or literal expression of a given type is given by the interpretation of the equivalent expression in Java. However, the precise behavior of some queries might vary depending on the native semantics of queries on the underlying datastore. For example, numeric precision and overflow, string collation, and integer division are permitted to depart from the semantics of the Java language.

This specification should not be interpreted to mandate an inefficient implementation of query language constructs in cases where the native behavior of the database varies from Java in such minor ways. That said, portability between Jakarta Query providers is maximized when their behavior is closest to the Java language.

Since an attribute of an entity may be null, a JQL expression may evaluate to a null value.

4.2. Lexical structure

Lexical analysis requires recognition of the following token types:

- keywords (reserved identifiers),
- regular identifiers,
- named and ordinal parameters,
- operators and punctuation characters,
- literal strings, and
- integer and decimal number literals.

4.2.1. Identifiers and keywords

An *identifier* is any legal Java identifier which is not a keyword. Identifiers are case-sensitive: `hello`, `Hello`, and `HELLO` are

distinct identifiers.

In the JQL grammar, identifiers are labelled with the `IDENTIFIER` token type.

The following identifiers are *keywords*: `select`, `update`, `set`, `delete`, `from`, `where`, `order`, `by`, `asc`, `desc`, `not`, `and`, `or`, `between`, `like`, `in`, `null`, `local`, `true`, `false`. In addition, every reserved identifier listed in section 4.4.1 of the Jakarta Persistence specification version 3.2 is also considered a reserved identifier. Keywords and other reserved identifiers are case-insensitive: `null`, `Null`, and `NULL` are three ways to write the same keyword.

Use of a reserved identifier as a regular identifier in JQL might be accepted by a given Jakarta Query provider, but such usage is not guaranteed to be portable between providers.

4.2.2. Parameters

A *named parameter* is a legal Java identifier prefixed with the `:` character, for example, `:name`.

An *ordinal parameter* is a decimal integer prefixed with the `?` character, for example, `?1`.

Ordinal parameters are numbered sequentially, starting with `?1`.

4.2.3. Operators and punctuation

The character sequences `+`, `-`, `*`, `/`, `||`, `=`, `<`, `>`, `<>`, `<=`, `>=` are *operators*.

The characters `(`, `)`, and `,` are *punctuation characters*.

When working with NoSQL databases, the support for arithmetic operations and support of parentheses for precedence might vary significantly:

Key-value databases

Arithmetic operations (`+`, `-`, `*`, `/`) are not supported. These databases are designed for simple key-based lookups and lack query capabilities for complex operations.

Wide-column databases

Arithmetic operations are not required to be supported. Some wide-column databases might offer limited support, which might require secondary indexing even for basic querying.

Document Databases

Support of arithmetic operations and support of parenthesis for precedence are not required, although databases typically offer these capabilities. Behavior and extent of support can vary significantly between providers.

Graph Databases

Support for arithmetic operations and parentheses for precedence are not required but is typically offered by databases. Behavior and extent of support can vary significantly between providers.

Due to the diversity of NoSQL database types and their querying capabilities, there is no guarantee that all NoSQL providers will support punctuation characters such as parentheses `(`, `)` for defining operation precedence. It is recommended to consult your NoSQL provider's documentation to confirm the supported query features and their behavior.

4.2.4. String literals

A *literal string* is a character sequence quoted using the character `'`.

A single literal `'` character may be included within a string literal by self-escaping it, that is, by writing `''`. For example, the string literal `'Furry''s theorem has nothing to do with furries.'` evaluates to the string `Furry's theorem has nothing to do with furries.`.

In the grammar, literal strings are labelled with the `STRING` token type.

4.2.5. Numeric literals

Numeric literals come in two flavors:

- any legal Java decimal literal of type `int` or `long` is an *integer literal*, and
- any legal Java literal of type `float` or `double` is a *decimal literal*.

In the grammar, integer and decimal literals are labelled with the `INTEGER` and `DOUBLE` token types respectively.

JQL does not require support for literals written in octal or hexadecimal.

4.2.6. Whitespace

The characters Space, Horizontal Tab, Line Feed, Form Feed, and Carriage Return are considered whitespace characters and make no contribution to the token stream.

As usual, token recognition is "greedy". Therefore, whitespace must be placed between two tokens when:

- a keyword directly follows an identifier or named parameter,
- an identifier directly follows a keyword or named parameter, or
- a numeric literal directly follows an identifier, keyword, or parameter.

4.3. Expressions

An expression is a sequence of tokens to which a Java type can be assigned, and which evaluates to a well-defined value when the query is executed. In JQL, expressions may be categorized as:

- literals,
- special values,
- parameters,
- enum literals,
- paths,
- function calls, and
- operator expressions.

4.3.1. Literal expressions

A string, integer, or decimal literal is assigned the type it would be assigned in Java. So, for example, `'Hello'` is assigned the type `java.lang.String`, `123` is assigned the type `int`, `1e4` is assigned the type `double`, and `1.23f` is assigned the type `float`.

The syntax for literal expressions is given by the `literal` grammar rule, and in the previous section titled Lexical structure.

When executed, a literal expression evaluates to its literal value.

4.3.2. Special values

The special values `true` and `false` are assigned the type `boolean`, and evaluate to their literal values.

The special values `local date`, `local time`, and `local datetime` are assigned the types `java.time.LocalDate`, `java.time.LocalTime`, and `java.time.LocalDateTime`, and evaluate to the current date and current datetime of the database server, respectively.

The syntax for special values is given by the `special_expression` grammar rule.

4.3.3. Parameter expressions

A parameter expression, with syntax given by `input_parameter`, is assigned the type of the repository method parameter it matches. For example, the parameter `:titlePattern` is assigned the type `java.lang.String`:

```
@Query("where title like :titlePattern")
List<Book> booksMatchingTitle(String titlePattern);
```

When executed, a parameter expression evaluates to the argument supplied to the parameter of the repository method.

Positional and named parameters must not be mixed in a single query.

4.3.4. Enum literals

An *enum literal expression* is a Java identifier, with syntax specified by `enum_literal`, and may only occur as the right operand of a `set` assignment or `=`/`<>` equality comparison. It is assigned the type of the left operand of the assignment or comparison. The type must be a Java `enum` type, and the identifier must be the name of an enumerated value of the `enum` type including the fully qualified Java enum class name. For example, `day <> java.time.DayOfWeek.MONDAY` is a legal comparison expression.

When executed, an enum expression evaluates to the named member of the Java `enum` type.

4.3.5. Path expressions

A *path expression* is a period-separated list of Java identifiers, with syntax specified by `state_field_path_expression`. Each identifier is interpreted as the name of an attribute of an entity or embeddable class. Each prefix of the list is assigned a Java type:

- the first element of the list is assigned the type of the named attribute of the entity being queried, and
- each subsequent element is assigned the type of the named attribute of the type assigned to the previous element.

The type of the whole path expression is the type of the last element of the list. For example, `pages` is assigned the type `int`, `address` is assigned the type `org.example.Address`, and `address.street` is assigned the type `java.lang.String`.

Typically, the last element of a path expression is assigned a basic type. Non-terminal path elements are usually assigned an embeddable type, if the element references an embedded attribute, or an entity type, if the element references an association attribute. However, since a Jakarta Query provider is not required to support embedded attributes or associations, a JQL implementation is not required to support compound path expressions.

When a path expression is executed, each element of the path is evaluated in turn:

- the first element of the path expression is evaluated in the context of a given record of the queried entity type, and evaluates to the value of the named entity attribute of the given record, and then
- each subsequent element is evaluated in the context of the result produced the previous element (typically, and embeddable class or associated entity class), and evaluates to the value of the named attribute of the result.

If any element of a path expression evaluates to a null value, the whole path expression evaluates to a null value.

4.3.6. Identifier expressions

An *identifier expression*, with syntax given by `id_expression`, is assigned the type of the unique identifier of the queried entity and evaluates to the unique identifier of a given record. An identifier expression is a synonym for a path expression with one element matching the identifier attribute of the queried entity type. An identifier expression may occur in the `select` clause, in the `order` clause, or as a scalar expression in the `where` clause.

4.3.7. Function calls

A *function call* is the name of a JQL function, followed by a parenthesized list of argument expressions, with syntax given by `function_expression`.

- The `abs()` function is assigned the type of its numeric argument, and evaluates to the absolute value of the numeric value to which its argument evaluates. Its argument must be of numeric type.
- The `length()` function is assigned the type `java.lang.Integer`, and evaluates to the length of string to which its argument evaluates. Its argument must be of type `java.lang.String`.
- The `lower()` function is assigned the type `java.lang.String`, and evaluates to the lowercase form of the string to which its argument evaluates. Its argument must be of type `java.lang.String`.
- The `upper()` function is assigned the type `java.lang.String`, and evaluates to the uppercase form of the string to which its argument evaluates. Its argument must be of type `java.lang.String`.
- The `left()` function is assigned the type `java.lang.String`, and evaluates to a prefix of the string to which its first argument evaluates. The length of the prefix is given by the integer value to which its second argument evaluates. The first argument must be of type `java.lang.String`, and the second argument must be of integral numeric type.
- The `right()` function is assigned the type `java.lang.String`, and evaluates to a suffix of the string to which its first argument evaluates. The length of the suffix is given by the integer value to which its second argument evaluates. The first argument must be of type `java.lang.String`, and the second argument must be of integral numeric type.

When any argument expression of any function call evaluates to a null value, the whole function call evaluates to null.

These functions cannot be emulated on every datastore. When a function cannot be reasonably emulated via the native query capabilities of the database, a JQL implementation is not required to provide the function.

If the JQL implementation does not support a standard function explicitly listed above, it must throw `UnsupportedOperationException` when the function name occurs in a query. Alternatively, the Jakarta Query provider is permitted to reject a repository method declaration at compilation time if its `@Query` annotation uses an unsupported function.

On the other hand, an implementation of JQL might provide additional built-in functions, and might even allow invocation of user-defined functions. Section 4.7 of the Jakarta Persistence specification defines a set of functions that all JPQL implementations are required to provide, including `concat`, `substring`, `trim`, `locate`, `ceiling`, `floor`, `exp`, `ln`, `mod`, `power`, `round`, `sign`, `sqrt`, `cast`, `extract`, `coalesce`, and `nullif`.

> JQL implementations are encouraged to support any of these functions which are reasonably implementable.

4.3.8. Operator expressions

The syntax of an *operator expression* is given by the `scalar_expression` rule. Within an operator expression, parentheses indicate grouping.

All binary infix operators are left-associative. The relative precedence, from highest to lowest precedence, is given by:

1. `*` and `/`,
2. `+` and `-`,
3. `||`.

The unary prefix operators `+` and `-` have higher precedence than the binary infix operators. Thus, `2 * -3 + 5` means `(2 * (-3)) + 5` and evaluates to `-1`.

The concatenation operator `||` is assigned the type `java.lang.String`. Its operand expressions must also be of type `java.lang.String`. When executed, a concatenation operator expression evaluates to a new string concatenating the strings to which its arguments evaluate.

The numeric operators `+`, `-`, `*`, and `/` have the same meaning for primitive numeric types they have in Java, and operator expression involving these operators are assigned the types they would be assigned in Java.

> As an exception, when the operands of `/` are both integers, a JQL implementation is not required to interpret the operator expression as integer division if that is not the native semantics of the database. However, portability is maximized when Jakarta Query providers *do* interpret such an expression as integer division.

The four numeric operators may also be applied to an operand of wrapper type, for example, to `java.lang.Integer` or `java.lang.Double`. In this case, the operator expression is assigned a wrapper type, and evaluates to a null value when either of its operands evaluates to a null value. When both operands are non-null, the semantics are identical to the semantics of an operator expression involving the corresponding primitive types.

The four numeric operators may also be applied to operands of type `java.math.BigInteger` or `java.math.BigDecimal`.

A numeric operator expression is evaluated according to the native semantics of the database. In translating an operator expression to the native query language of the database, a Jakarta Query provider is encouraged, but not required, to apply reasonable transformations so that evaluation of the expression more closely mimics the semantics of the Java language.

4.3.9. Numeric types and numeric type promotion

The type assigned to an operator expression depends on the types of its operand expression, which need not be identical. The rules for numeric promotion are given in section 4.7 of the Jakarta Persistence specification version 3.2:

> - *If there is an operand of type* `Double` *or* `double`*, the expression is of type* `Double`*;*
> - *otherwise, if there is an operand of type* `Float` *or* `float`*, the expression is of type* `Float`*;*
> - *otherwise, if there is an operand of type* `BigDecimal`*, the expression is of type* `BigDecimal`*;*
> - *otherwise, if there is an operand of type* `BigInteger`*, the expression is of type* `BigInteger`*, unless the operator is* `/` *(division), in which case the expression type is not defined here;*

> - *otherwise, if there is an operand of type* `Long` *or* `long`*, the expression is of type* `Long`*, unless the operator is* `/` *(division), in which case the expression type is not defined here;*
> - *otherwise, if there is an operand of integral type, the expression is of type* `Integer`*, unless the operator is* `/` *(division), in which case the expression type is not defined here.*

4.4. Conditional expressions

A *conditional expression* is a sequence of tokens which specifies a condition which, for a given record, might be *satisfied* or *unsatisfied*. Unlike the scalar Expressions defined in the previous section, a conditional expression is not considered to have a well-defined type.

JPQL defines the result of a conditional expression in terms of ternary logic. JQL does not specify that a conditional expression evaluates to well-defined value, only the effect of the conditional expression when it is used as a restriction. The "value" of a conditional expression is not considered observable by the application program.

Conditional expressions may be categorized as:

- `null` comparisons,
- `in` expressions,
- `between` expressions,
- `like` expressions,
- equality and inequality operator expressions, and
- logical operator expressions.

The syntax for conditional expressions is given by the `conditional_expression` rule. Within a conditional expression, parentheses indicate grouping.

4.4.1. Null comparisons

A `null` comparison, with syntax given by `null_comparison_expression` is satisfied when:

- the `not` keyword is missing, and its operand evaluates to a null value, or
- the `not` keyword occurs, and its operand evaluates to any non-null value.

4.4.2. In expressions

An `in` expression, with syntax given by `in_expression` is satisfied when its leftmost operand evaluates to a non-null value, and:

- the `not` keyword is missing, and any one of its parenthesized operands evaluates to the same value as its leftmost operand, or
- the `not` keyword occurs, and none of its parenthesized operands evaluate to the same value as its leftmost operand.

All operands must have the same type.

4.4.3. Between expressions

A `between` expression, with syntax given by `between_expression` is satisfied when its operands all evaluate to non-null values, and, if the `not` keyword is missing, its left operand evaluates to a value which is:

- larger than or equal to the value taken by its middle operand, and
- smaller than or equal to the value taken by its right operand.

Or, if the `not` keyword occurs, the left operand must evaluate to a value which is:

- strictly smaller than to the value taken by its middle operand, or
- strictly larger than the value taken by its right operand.

All three operands must have the same type.

4.4.4. Like expressions

A `like` expression is satisfied when its left operand evaluates to a non-null value and:

- the `not` keyword is missing, and this value matches the pattern, or
- the `not` keyword occurs, and the value does not match the pattern.

The left operand must have type `java.lang.String`.

Within the pattern, `_` matches any single character, and `%` matches any sequence of characters.

4.4.5. Equality and inequality operators

The equality and inequality operators are `=`, `<>`, `<`, `>`, `<=`, `>=`.

- For primitive types, these operators have the same meaning they have in Java, except for `<>` which has the same meaning that `!=` has in Java. Such an operator expression is satisfied when the equivalent operator expression would evaluate to `true` in Java.
- For wrapper types, these operators are satisfied if both operands evaluate to non-null values, and the equivalent operator expression involving primitives would be satisfied.
- For other types, these operators are evaluated according to the native semantics of the database.

The operands of an equality or inequality operator must have the same type.

Portability is maximized when Jakarta Query providers interpret equality and inequality operators in a manner consistent with the implementation of `Object.equals()` or `Comparable.compareTo()` for the assigned Java type.

When using NoSQL databases, there are limitations to the support of equality and inequality operators:

1. **Key-Value Databases**: Support for the equality restriction on the key attribute is required. The key attribute is defined by the annotation `jakarta.nosql.Id`. Key-value databases are not required to support any other restrictions.
2. **Wide-Column Databases**: Support for equality restriction and the inequality restriction on the `Id` attribute is required. Support for restrictions on other entity attributes is not required. These operations typically work only with the `Id` by default but might be compatible for other entity attributes if secondary indexes are configured in the database schema.
3. **Graph and Document Databases**: Support for all equality and inequality operators is required.

4.4.6. Ordering

Every basic type can, in principle, be equipped with a total order. An order for a type determines the result of

inequality comparisons, and the effect of the Order clause.

For numeric types, and for date, time, and datetime types, the total order is unique and completely determined by the semantics of the type. JQL implementations must sort these types according to their natural order, that is, the order in JQL must agree with the order defined by Java.

Boolean values must be ordered so that `false < true` is satisfied.

For other types, there is at least some freedom in the choice of order. Usually, the order is determined by the native semantics of the database. Note that:

- Textual data is represented in JQL as the type `java.lang.String`. Strings are in general ordered lexicographically, but the ordering also depends on the character set and collation used by the database server. Applications must not assume that the order agrees with the `compareTo()` method of `java.lang.String`. In evaluating an inequality involving string operands, an implementation of JQL is not required to emulate Java collation.
- Binary data is represented in JQL as the type `byte[]`. Binary data is in general ordered lexicographically with respect to the constituent bytes. However, since this ordering is rarely meaningful, this specification does not require implementations of JQL to respect it.
- This specification does not define an order for the sorting of Java `enum` values, which is provider-dependent. A programming model for entity classes might allow control over the order of `enum` values. For example, Jakarta Persistence allows this via the `@Enumerated` annotation.
- This specification does not define an order for UUID values, which is provider-dependent.

When using NoSQL databases, sorting support varies by database type:

Key-value databases

Sorting of results is not supported.

Wide-column databases

Support for sorting of results is not required. In general, sorting is not natively supported. When sorting is available, it is typically limited to:

- The key attribute, defined by an annotation such as `jakarta.nosql.Id`.
- Fields that are indexed as secondary indexes.

Graph and document databases

Support for sorting by a single entity attribute is required. Support for compound sorting (sorting by multiple entity attributes) is not required and may vary due to:

- Potential instability with tied values, where sorting for equivalent values may differ across queries.
- Schema flexibility and mixed data types.
- Dependence on indexes and internal storage order, requiring proper indexing to ensure predictable sorting.
- The distributed nature of sharded clusters, where sorting across shards may introduce additional complexity.

4.4.7. Logical operators

The logical operators are `and`, `or`, and `not`.

- An `and` operator expression is satisfied if and only if both its operands are satisfied.

- An `or` operator expression is satisfied if at least one of its operands is satisfied.
- A `not` operator expression is never satisfied if its operand *is* satisfied.

This specification leaves undefined the interpretation of the `not` operator when its operand *is not* satisfied.

A compliant implementation of JQL might feature SQL/JPQL-style ternary logic, where `not n > 0` is an unsatisfied logical expression when `n` evaluates to null, or it might feature binary logic where the same expression is considered satisfied. Application programmers should take great care when using the `not` operator with scalar expressions involving `null` values.

Syntactically, logical operators are parsed with lower precedence than equality and inequality operators and other conditional expressions listed above. The `not` operator has higher precedence than `and` and `or`. The `and` operator has higher precedence than `or`.

When using NoSQL databases, the support for restrictions varies depending on the database type:

Key-value databases

Support for the equality restriction is required for the `Id` attribute. There is no requirement to support other types of restrictions or restrictions on other entity attributes.

Wide-column databases

Wide-column databases are not required to support the `AND` operator or the `OR` operator. Restrictions must be supported for the key attribute that is annotated with `jakarta.nosql.Id`. Support for restrictions on other attributes is not required. Typically they can be used if they are indexed as secondary indexes, although support varies by database provider.

Graph and document databases

The `AND` and `OR` operators and all of the restrictions described in this section must be supported. Precedence between `AND` and `OR` operators is not guaranteed and may vary significantly based on the NoSQL provider.

4.5. Clauses

Each JQL statement is built from a sequence of *clauses*. The beginning of a clause is identified by a keyword: `from`, `where`, `select`, `set`, or `order`.

There is a logical ordering of clauses, reflecting the order in which their effect must be computed by the datastore:

1. `from`
2. `where`,
3. `select` or `set`,
4. `order`.

The interpretation and effect of each clause in this list is influenced by clauses occurring earlier in the list, but not by clauses occurring later in the list.

4.5.1. From clause

The `from` clause, with syntax given by `from_clause`, specifies an *entity name* which identifies the queried entity. Path expressions occurring in later clauses are interpreted with respect to this entity. That is, the first element of each path expression in the query must be a persistent attribute of the entity named in the `from` clause. The entity name is a Java

identifier, usually the unqualified name of the entity class, as specified in [Entity names].

The syntax of the `update` statement is irregular, with the `from` keyword implied. That is, the syntax *should* be `update from Entity`, but for historical reasons it is simply `update Entity`.

The `from` clause is optional in `select` statements. When it is missing, the queried entity is determined by the return type of the repository method, or, if the return type is not an entity type, by the primary entity type of the repository.

For example, this repository method declaration:

```
@Query("where title like :title")
List<Book> booksByType(String title);
```

is equivalent to:

```
@Query("from Book where title like :title")
List<Book> booksByType(String title);
```

4.5.2. Where clause

The `where` clause, with syntax given by `where_clause`, specifies a conditional expression used to restrict the records returned, deleted, or updated by the query. Only records for which the conditional expression is satisfied are returned, deleted, or updated.

The `where` clause is always optional. When it is missing, there is no restriction, and all records of the queried entity type are returned, deleted, or updated.

4.5.3. Select clause

The `select` clause, with syntax given by `select_clause`, specifies one or more path expressions which determine the values returned by the query. Each path expression is evaluated for each record which satisfies the restriction imposed by the `where` clause, as specified in Path expressions, and a tuple containing the resulting values is added to the query results.

When a `select` clause contains more than one item, the query return type must be a Java `record` type, and the elements of the tuple are repackaged as an instance of the query return type by calling a constructor of the `record`, passing the elements in the same order they occur in the `select` list. When the `select` clause contains only one item, the query directly returns the values of the path expression.

Alternatively, the `select` clause may contain either:

- a single `count(this)` aggregate expression, which evaluates to the number of records which satisfy the restriction, or
- a single identifier expression, which evaluates to the unique identifier of each record.

```
A query beginning with `select count(this)` always returns a single result of type `Long`, no matter how many
records satisfy the conditional expression in the `where` clause.
```

If a datastore does not natively provide the ability to count query results, the Jakarta Query provider is strongly encouraged, but not required, to implement this operation by counting the query results in Java.

If the JQL implementation does not support `count(this)`, it must throw `UnsupportedOperationException` when this aggregate

expression occurs in a query. Alternatively, the Jakarta Query provider is permitted to reject a repository method declaration at compilation time if its `@Query` annotation uses the unsupported aggregate expression.

The `select` clause is optional in `select` statements. When it is missing, the query returns the queried entity.

When working with NoSQL databases, the `select` clause behavior may vary depending on the database structure and capabilities:

Key-value databases

These databases generally do not support `select` clauses beyond retrieving values by their keys. Support for complex path expressions and aggregate functions like `count(this)` is not required.

Wide-column databases

The ability to use a `select` clause may depend on the presence of secondary indexes. Without secondary indexes, selection is often restricted to key-based operations. Support for `count(this)` is not required.

Graph and document databases

Support for flexible `select` clauses, including path expressions and aggregate functions like `count(this)` is required. Performance might vary based on the size and indexing of the dataset.

For `count(this)` in particular, if the NoSQL datastore does not natively support counting query results, the Jakarta Query provider is encouraged to implement this operation in Java. However, providers are not required to do so. If `count(this)` is unsupported, an `UnsupportedOperationException` must be thrown during query execution, or repository methods using this expression may be rejected at compilation time.

It is advisable to review your NoSQL provider's documentation to confirm the support and performance implications of `select` clauses and aggregate functions in your queries.

4.5.4. Set clause

The `set` clause, with syntax given by `set_clause`, specifies a list of updates to attributes of the queried entity. For each record which satisfies the restriction imposed by the `where` clause, and for each element of the list, the scalar expression is evaluated and assigned to the entity attribute identified by the path expression.

4.5.5. Order clause

The `order` clause (or `order by` clause), with syntax given by `orderby_clause`, specifies a lexicographic order for the query results, that is, a list of entity attributes used to sort the records which satisfy the restriction imposed by the `where` clause. The keywords `asc` and `desc` specify that a given attribute should be sorted in ascending or descending order respectively; when neither is specified, ascending order is the default.

An implementation of JQL is not required to support sorting by entity attributes which are not returned by the query. If a query returns the queried entity, then any sortable attribute of the queried entity may occur in the `order` clause. Otherwise, if the query has an explicit `select` clause, a provider might require that an attribute which occurs in the `order` also occurs in the `select`.

Entity attributes occurring earlier in the `order by` clause take precedence. That is, an attribute occurring later in the `order by` clause is only used to resolve "ties" between records which cannot be unambiguously ordered using only earlier attributes.

This specification does not define how null values are ordered with respect to non-null values. The ordering of null values may vary between data stores and between Jakarta Query providers.

The `order` clause is always optional. When it is missing, and when no sort criteria are given as arguments to a parameter of the repository method, the order of the query results is undefined, and might not be deterministic.

If a datastore does not natively provide the ability to sort query results, the Jakarta Query provider is strongly encouraged, but not required, to sort the query results in Java before returning the results to the client.

If the Jakarta Query provider cannot satisfy a request for sorted query results, it must throw `DataException`.

4.6. Statements

Finally, there are three kinds of *statement*:

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

The clauses which can appear in a statement are given by the grammar for each kind of statement.

4.6.1. Select statements

A `select` statement, with syntax given by `select_statement`, returns data to the client. For each record which satisfies the restriction imposed by the `where` clause, a result is returned containing the value obtained by evaluating the path expression in the `select` clause. Alternatively, for the case of `select count(this)`, the query returns the number of records which satisfied the restriction.

4.6.2. Update statements

An `update` statement, with syntax given by `update_statement`, updates each record which satisfies the restriction imposed by the `where` clause, and returns the number of updated records to the client.

If a NoSQL database is not capable of conditional updates or cannot determine the number of matching records reliably for an `update` operation that returns an `int` or `long`, the `update` operation must throw an `UnsupportedOperationException`.

Additionally, in databases with **append-only semantics**—such as many time-series and wide-column databases—the `update` operation may behave more like an `insert`, and repeated updates to the same record might not overwrite previous values.

4.6.3. Delete statements

A `delete` statement, with syntax given by `delete_statement`, deletes each record which satisfies the restriction imposed by the `where` clause, and returns the number of deleted records to the client.

If a NoSQL database is not capable of the execution of conditional deletes or cannot determine the number of deleted records reliably for a `delete` operation that returns an `int` or `long`, the `delete` operation must throw an `UnsupportedOperationException`.

4.7. Syntax

The following grammar defines the syntax of JQL, via ANTLR4-style BNF.

```
grammar JQL;

statement : select_statement | update_statement | delete_statement;

select_statement : select_clause? from_clause? where_clause? orderby_clause?;
update_statement : 'UPDATE' entity_name set_clause where_clause?;
delete_statement : 'DELETE' from_clause where_clause?;

from_clause : 'FROM' entity_name;

where_clause : 'WHERE' conditional_expression;

set_clause : 'SET' update_item (',' update_item)*;
update_item : state_field_path_expression '=' (scalar_expression | 'NULL');

select_clause : 'SELECT' (select_item | select_items);
select_item
    : state_field_path_expression
    | id_expression
    | aggregate_expression
    ;
select_items
    : state_field_path_expression (',' state_field_path_expression)+
    ;

orderby_clause : 'ORDER' 'BY' orderby_item (',' orderby_item)*;
orderby_item : (state_field_path_expression | id_expression) ('ASC' | 'DESC');

conditional_expression
    // highest to lowest precedence
    : '(' conditional_expression ')'
    | null_comparison_expression
    | in_expression
    | between_expression
    | like_expression
    | comparison_expression
    | 'NOT' conditional_expression
    | conditional_expression 'AND' conditional_expression
    | conditional_expression 'OR' conditional_expression
    ;

comparison_expression : scalar_expression ('=' | '>' | '>=' | '<' | '<=' | '<>') scalar_expression;
between_expression : scalar_expression 'NOT'? 'BETWEEN' scalar_expression 'AND' scalar_expression;
like_expression : scalar_expression 'NOT'? 'LIKE' STRING;

in_expression : state_field_path_expression 'NOT'? 'IN' '(' in_item (',' in_item)* ')';
in_item : literal | enum_literal | input_parameter;

null_comparison_expression : state_field_path_expression 'IS' 'NOT'? 'NULL';

scalar_expression
    // highest to lowest precedence
    : '(' scalar_expression ')'
    | primary_expression
    | ('+' | '-') scalar_expression
    | scalar_expression ('*' | '/') scalar_expression
    | scalar_expression ('+' | '-') scalar_expression
    | scalar_expression '||' scalar_expression
    ;

primary_expression
    : function_expression
```

```
    | special_expression
    | id_expression
    | state_field_path_expression
    | enum_literal
    | input_parameter
    | literal
    ;

id_expression : 'ID' '(' 'THIS' ')' ;

aggregate_expression : 'COUNT' '(' 'THIS' ')';

function_expression
    : 'ABS' '(' scalar_expression ')'
    | 'LENGTH' '(' scalar_expression ')'
    | 'LOWER' '(' scalar_expression ')'
    | 'UPPER' '(' scalar_expression ')'
    | 'LEFT' '(' scalar_expression ',' scalar_expression ')'
    | 'RIGHT' '(' scalar_expression ',' scalar_expression ')'
    ;

special_expression
    : 'LOCAL' 'DATE'
    | 'LOCAL' 'DATETIME'
    | 'LOCAL' 'TIME'
    | 'TRUE'
    | 'FALSE'
    ;

state_field_path_expression : IDENTIFIER ('.' IDENTIFIER)*;

entity_name : IDENTIFIER; // no ambiguity

enum_literal : IDENTIFIER ('.' IDENTIFIER)*; // ambiguity with state_field_path_expression resolvable semantically

input_parameter : ':' IDENTIFIER | '?' INTEGER;

literal : STRING | INTEGER | DOUBLE;
```