JAKARTA EE

Jakarta Query

1.0-M1, October 11, 2025: Draft

Table of Contents

Specification: Jakarta Query

Version: 1.0-M1

Status: Draft

Release: October 11, 2025

Chapter 1. Introduction

Jakarta Query defines an object-oriented query language designed for use with Jakarta Persistence, Jakarta Data, Jakarta NoSQL, and with any other similar persistence technology in any object-oriented programming language. Jakarta Query specifies:

- 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.

An object-oriented query language is in many ways very similar to a graph-based query language like, for example, SPARQL, but differs from such query languages by assuming a typed representation of the data native to the object-oriented programming language.

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 data stores.

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.

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:

- 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.

This document does not define how constructs 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

Italics are used to highlight a word or phrase which is being defined by the text in which it occurs.

Bold text is occasionally used to draw attention to important words or phrases.

`Monospace` font indicates a keyword or fragment of text from the query language itself, or from the grammar for the language. When a keyword or name of a grammar rule is not part of the core language, the keyword or rule name is `highlighted`.

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

An informal but readable BNF-like metalanguage is used to illustrate the syntax of some basic language elements in Basic operations. Here, repetition is indicated by a comma-separated list terminated with an ellipsis, and production is indicated by a label enclosed in brackets; all other tokens are interpreted literally. Code fragments written in this metalanguage are never prescriptive.

Admonitions are used to set apart:

- explanatory text which is not prescriptive or not part of the definition of the language, and
- information relating to compatibility between implementations of Jakarta Query, and to differences between the core language and the extended language.

Throughout this specification, the phrase *"an implementation for Java"* means an implementation of Jakarta Query which targets clients written in the Java programming language and running on the Java Virtual Machine. It does not imply that the implementation itself is entirely in Java. This specification places certain requirements on implementations for Java which are relaxed when an implementation targets clients in a different programming 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.

An implementation of this specification in Java typically supports at least the following atomic types: Java primitive types, `String`, `LocalDate`, `LocalDateTime`, `LocalTime`, `Year`, and `Instant`, `java.util.UUID`, `java.math.BigInteger` and `java.math.BigDecimal`, `byte[]`, and `enum` types.

An atomic value might be *null*. Null values are conventionally used to represent missing or unknown information.[1] A null value is considered distinct from and distinguishable from every non-null value of the same type. The relationship between distinctness in this sense and equality is not defined by this specification, and the behavior of some expressions involving null values varies between implementations of Jakarta Query.

An implementation of Jakarta Query is permitted to treat null values as equal, so that, in the sense defined by Conditional expressions, the comparison expression `null = null` is satisfied while `null <> null` is unsatisfied. Such implementations are typically based on binary logic. On the other hand, an implementation is permitted to treat null values as unequal, so that both `null = null` and `null <> null` are unsatisfied. Such implementations are typically based on SQL-style ternary logic. Jakarta Persistence requires the use of ternary logic and treats null values as unequal. Jakarta Data permits either treatment.

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.

As usual:

- A set never contains two identical elements; a map never contains two identical keys.
- On the other hand, a list may contain identical elements at different positions within the list[2]; a map may contain identical values when their keys are non-identical.

2.3. Structures and records

A *structure* is a finite set of labeled 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 labeled elements have the same type.

A structure has no identity independent of the labeled 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 labeled 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.[3]

- 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 `_`.[4]

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. [5]

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 labeled `y` in `Y`, there is a corresponding type labeled `y` in `X` and either:

- the two types with label `y` are identical, or
- the type labeled `y` in `Y` is an entity type `T`, the type labeled `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 labeled `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* labeled `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`, labeled `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.

Circularity typically arises in implementations which support *associations* between entity types. An association is a labeled element of an entity or embeddable type which is itself an entity type. This specification does not require that implementations allow associations between entities, and the core language does not feature constructs for querying associations. In particular, an implementation of Jakarta Query which targets a non-relational database might not feature any support for associations. On the other hand, the extended language defines constructs like joins and typecasts which are intended for use in implementations which do support associations. In particular, association mapping is a fundamental feature of Jakarta Persistence, and circularity is completely normal in implementations based on object/relational mapping.

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.

2.7. Mapping to Java

When Jakarta Query is used from within the Java programming language:

- An entity or embeddable type typically corresponds to a Java class or record type, and the labeled elements of an entity or embeddable typically correspond to fields or properties of the class or record type.
- An atomic value type typically corresponds to a Java primitive type, class, or enumerated type.
- Every value expression in a query is assigned a Java type. Conceptually — but not literally — the expression evaluates to a (possibly null) instance of that type.

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 implementations of Jakarta Query is maximized when their behavior is closest to the Java language.

2.8. Paths

Consider an arbitrary root structure constructed recursively using only atomic values and structures (and no collections). Such a structure may be viewed as a directed tree, where vertices are structures and atomic values, and edges are labeled structure elements. Then it is possible to uniquely assign a compound label to any given element of any structure in the tree by:

1. tracing a directed path from the given element to the root structure, and
2. collecting the labels of each element visited along the path.

Conventionally, we write such a compound label in reverse order, beginning with the label of the element belonging to the root structure and ending with the label of the most nested element, and we separate labels with periods.

For example, given the structure:

```
{
    author: {
        name: "Iain M. Banks",
        born: {
            day: 16, month: { name: 'February', number: 2 }, year: 1954
        },
        died: {
            day: 9, month: { name: 'June', number: 6 }, year: 2013
        }
    }
}
```

The compound label `author.born.month.name` refers to the element `name: 'February'`.

It is not possible to uniquely assign compound labels to every element belonging to an arbitrary structure, since:

- a generic structure contains sets, whose elements cannot be assigned labels[6], and
- as discussed above in Circularity, a database is not required to contain only trees.

However, we may use such labelling within any subtree that does not contain a collection, though in principle the compound labels might not be unique.

[1] The null value is also indispensable when evaluating path expressions in queries involving left joins and when evaluating treated path expressions.

[2] When discussing query result lists, we sometimes say that identical elements belonging to a list are *duplicates*.

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

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

[5] 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.

[6] It would be possible, of course, to assign labels to elements of a list or map.

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**,
- **duplicate elimination**, 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, which we sometimes call its *shape*, according to the rules specified above. A result list may contain two or more structures which are identical in the sense of Structures and records.

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.

Finally, the binary operations **union**, **intersection**, and **complementation** may be used to join the outputs of multiple pipelines.

3.1. The root entity

Every query begins with a *root entity* (unless it is a correlated subquery). 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 labeled element: the record, labeled by the alias `x`.

That is, for each record *r* in the database, the result list has a structure *s* containing *r* labeled 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 labeled element to a result list. Evaluating a sequence of joins produces a result list in which each structure contains a labeled 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 labeled elements of *r* together with the record *y* labeled 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 labeled elements, one for each entity, each labeled 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 Conditional expressions.

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 or collections

Instead of a 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 labeled elements of *r* together with the structure *s* labeled 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 labeled elements of *r* together with the structure *e* labeled by the alias `y`.

The previous example evaluates to a list containing a structure for each `Author` of each `Book`. The structure contains two labeled elements, one for each entity, each labeled 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.

A join to a structure or collection may be immediately followed by a restriction.

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

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

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.

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 labeled elements of *r* together with the structure *s* labeled 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 labeled elements of *r* together with the structure *e* labeled 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 labeled elements of *r*.

Just like regular joins, a left join may be followed by a restriction:

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

In this case, however, the restriction is only evaluated for elements of the result list which contain structures labeled by the alias `y`. Any element of the result list of the join which does not contain a structure labeled by the alias `y` is taken to satisfy the restriction, even when the predicate would not be satisfied by that element.

3.2.4. Fetch joins

A fetch join is a hint to the Jakarta Query implementation regarding which data in the query result set the application program will access. A fetch join does not affect the shape (type) of the query result set, and therefore does not introduce a label. Otherwise, the syntax is similar to a left join:

```
[from] left outer join fetch [path]
```

```
[from] left join fetch [path]
```

The sister specifications of this specification assign semantics to this syntax.

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 Conditional expressions. 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 calls.

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 Value expressions.

1. For each structure *r* of the result list of the operation on which the ordering operation 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 Value 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 Value expressions. For each value expression with label `x` in the given `select` list, *r′* contains a element labeled `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 calls.

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.

Alternatively, if every value expression in the `select` list is an aggregate function, and if the query does *not* have a `group by` clause, then aggregation over all elements of the result list is implied.[2] Such a query produces a result list with a single element.

A query with no `group by` clause may not mix aggregate function expressions with other value expressions.

3.6. Duplicate elimination

The `distinct` keyword, placed after `select`, specifies that duplicate structures should be eliminated from the result list. Two structures are considered duplicates if they are identical in the sense defined in Structures and records.

Suppose the result list of a projection contains $n \geq 0$ identical copies of a structure r. Then duplicate elimination produces a result list containing (exactly one instance of) the structure r if and only $n \geq 1$. Thus, any two structures belonging to the final result list are distinct (non-identical).

Duplicate elimination does not change the shape (type) of the result list.

3.7. 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. Some general principles for ordering of atomic types are established in Natural order.

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
```

3.8. Union, intersection, and complement

The results of two queries may be combined via union, intersection, or complementation of their result lists viewed as sets. The syntax of a *union*, *intersection*, or *complement* is, respectively:

```
[query] union [query]
```

```
[query] intersect [query]
```

```
[query] except [query]
```

where each `[query]` is a complete query pipeline as described above.

The `union`, `intersect`, and `except` operators are operators on sets. That is, they:

- produce result lists with distinct elements, and
- are not required to preserve any element ordering which might exist in the result lists being combined.

Suppose a `union`, `intersect`, or `except` operator is applied to the result lists l_L and l_R, with l_L occurring to the left. The final result list contains a structure *r* if and only if:

1. the operator is `union`, and *r* occurs in either l_L or l_R,
2. the operator is `intersect`, and *r* occurs in both l_L or l_R, or
3. the operator is `except`, and *r* occurs in l_L but not in l_R.

The semantics of union, intersection, or complementation may be modified by the `all` keyword:

```
[query] union all [query]
```

```
[query] intersect all [query]
```

```
[query] except all [query]
```

When `all` occurs, the operation preserves duplicate results from the argument result lists and so the final result list might contain duplicate elements.

Suppose a `union all`, `intersect all`, or `except all` operator is applied to the result lists l_L and l_R, with l_L occurring to the left. Given a structure r which occurs n_L times in l_L and n_R times in l_R:

1. r occurs $n_L + n_R$ times in the final result list if the operator is `union all`,
2. r occurs $\min(n_L, n_R)$ times in the final result list if the operator is `intersect all`, or
3. r occurs $\max(0, n_L - n_R)$ times in the final result list if the operator is `except all`.

In principle, a union may act on result lists of distinct type, producing a result list containing structures of heterogeneous type. This specification requires union, intersection, or complementation only for lists of identical type because many data stores are not capable of unions that produce lists of heterogeneous type. Some implementations allow unions of lists containing entities of distinct type when the entity types are related by inheritance, but that functionality is not required by this specification.

3.9. Subqueries

A *subquery* is a query which occurs as an expression in another *outer query*. Subqueries have a restricted syntax compared to root queries, and, in particular, every subquery must have a projection with exactly one value expression.

This specification defines the behavior of subqueries occurring within restrictions in the `where` clause or `having` clause. Some implementations of Jakarta Query allow subqueries to occur elsewhere in the query, but this functionality falls outside the scope of this specification.

A subquery may have expressions which involve aliases declared by any outer query. A query with such expressions is called *correlated*.

- A subquery which is not correlated may be executed independently of the containing query, and its result set is fixed for all rows of the outer query.
- A correlated subquery must be evaluated once for each given element of the result list to which the restricted is applied. A correlated subquery does not have a root entity, and every element of the `from` clause is treated as a join. When the correlated subquery is executed, the first join in the pipeline is applied to a result list containing only the given element.

A subquery must not have an Ordering.

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

[2] That is, the query functions as if it had an empty `group by` clause with no value expressions, so that every element of the result list was assigned a grouping tuple of length zero, resulting in a result list containing a single nested list.

Chapter 4. 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.

4.1. 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 grammar, identifiers are labeled with the `IDENTIFIER` token type.

The following identifiers are *keywords*: `abs`, `all`, `and`, `any`, `as`, `asc`, `avg`, `between`, `both`, `by`, `case`, `ceiling`, `class`, `coalesce`, `concat`, `count`, `current_date`, `current_time`, `current_timestamp`, `delete`, `desc`, `distinct`, `else`, `empty`, `end`, `entry`, `escape`, `except`, `exists`, `exp`, `extract`, `false`, `fetch`, `first`, `floor`, `from`, `function`, `group`, `having`, `in`, `index`, `inner`, `intersect`, `is`, `join`, `key`, `leading`, `last`, `left`, `length`, `like`, `local`, `ln`, `locate`, `lower`, `max`, `member`, `min`, `mod`, `new`, `not`, `null`, `nulls`, `nullif`, `object`, `of`, `on`, `or`, `order`, `outer`, `position`, `power`, `replace`, `right`, `round`, `select`, `set`, `sign`, `size`, `some`, `sqrt`, `substring`, `sum`, `then`, `trailing`, `treat`, `trim`, `true`, `type`, `union`, `update`, `upper`, `value`, `when`, `where`.[1].

TODO: function names don't really need to be keywords!

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 a query might be accepted by a given Jakarta Query provider, but such usage is not guaranteed to be portable between providers.

4.2. 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`.

4.3. Operators and punctuation

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

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

4.4. 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 labeled with the `STRING` token type.

4.5. Numeric literals

In the core language, 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*.

A suffix `L`, `D`, or `F` may be used to indicate a specific numeric type:

- `L` means a 64-bit integer,
- `D` means a 64-bit floating-point value, and
- `F` means a 32-bit floating-point value

The suffix is not case-sensitive.

The literal numeric value preceding the suffix must conform to the rules for Java numeric literals established by the Java Language Specification.

When the suffix is absent:

- a literal with neither exponent nor decimal point is interpreted as a 32-bit integer value, and
- a literal with either an exponent or decimal point is interpreted as a 64-bit floating-point value.

An implementation of Jakarta Query which targets a language other than Java is not required to respect the suffix in a numeric literal.

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

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

In the extended language, arbitrary-precision numeric literals are also provided:

- an arbitrary-precision integer literal follows the format of a Java integer literal, but with the explicit suffix `BI`, and with no limit on the number of decimal digits, and
- an arbitrary-precision decimal literal follows the format of a Java floating-point literal, but with the explicit suffix `BD`, and with no limit on the number of decimal digits.

4.6. Single-character literals

The `trim_character` and `escape_character` rules allow specification of a single literal character quoted using the `'` character.

In the grammar, such single-character literals are labeled with the `CHARACTER` token type.

4.7. 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.

[1] Jakarta Persistence reserves the following additional keywords for future use: `BIT_LENGTH`, `CHAR_LENGTH`, `CHARACTER_LENGTH`, and `UNKNOWN`.

Chapter 5. Expressions

5.1. Value expressions

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

- literal atomic values, enum literals, and special values,
- parameters,
- paths,
- function calls and aggregate function calls,
- typecasts, type expressions, and literal entity types,
- case expressions,
- operator expressions, and
- Subquery expressions.

5.1.1. Literal expressions

The syntax for literal expressions is given by the rules `string_literal` and `numeric_literal`, and in the previous section titled Lexical structure.

In an implementation for Java, 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`. In the extended language:

- arbitrary-precision integer literals (with suffix `BI`) are assigned the type `java.math.BigInteger`, and
- arbitrary-precision decimal literals (with suffix `BD`) are assigned the type `java.math.BigDecimal`.

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

5.1.2. Special values

The syntax for *special values* is given by the rules `special_boolean_expression` and `special_datetime_expression`.

In an implementation of Jakarta Query for Java:

- the special values `true` and `false` are assigned the type `boolean`, and
- the special values `local date`, `local time`, and `local datetime` are assigned the types `java.time.LocalDate`, `java.time.LocalTime`, and `java.time.LocalDateTime`.

The expression `true` and `false` evaluate to their literal values. The expressions `local date`, `local time`, and `local datetime` evaluate to the current date, current time, and current datetime of the database server, respectively.

5.1.3. Parameter expressions

A *parameter expression*, with syntax given by `input_parameter`, is assigned a type and value either:

1. at runtime, via invocation of some API (for example, `setParameter()` in Jakarta Persistence), or
2. at compile type, via static analysis (for example, by inspecting the parameters of a Jakarta Data repository method).

This specification does not define how arguments are assigned to parameter expressions.

When executed, a parameter expression evaluates to the argument assigned to the parameter.

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

5.1.4. Enum literals

An *enum literal expression* is a sequence of identifiers, with syntax specified by `enum_literal`, and must refer to a member of some sort of enumerated type. An enum literal may only occur as:

- the right operand of an `update_item` belonging to a `set` assignment,
- the right operand of an `=` or `<>` equality comparison, or
- an `in_item` element of an `in_item_list` occurring as the right operand of the `in` operator.

In each case, the enum literal expression is assigned the type of the left operand of the assignment, comparison, or `in` operator.

In an implementation for Java, the type assigned 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, the enum literal expression evaluates to the named member of the enumerated type.

TODO allow `day <> MONDAY`, since the `enum` type can be inferred.

5.1.5. Entity type literals

In the extended language, an *entity type literal*, with syntax given by `entity_type_literal` is just the name of an entity type, in the sense given in Entities and embeddable types. When executed, it evaluates to a value which represents the named entity type.

An implementation for Java assigns the type `java.lang.Class<E>` to an entity type literal, where `E` is the Java class mapping the named entity type. When executed, the literal evaluates to an instance of `Class` representing the Java class `E`.

5.1.6. Path expressions

A *path expression* has syntax specified by:

- `simple_path_expression` in the core language, and
- `single_valued_path_expression`, `atomic_valued_path_expression`, `embeddable_valued_path_expression`, `entity_valued_path_expression`, `collection_valued_path_expression`, or `joinable_path_expression` in the extended language.

A `simple_path_expression` or `joinable_path_expression` is simply a period-separated list of identifiers. Any other sort of path expression might begin with:

- an identifier,
- a treated path expression, or
- a map key or value expression.

The first element of the path expression is called the *root element* of the path.

A path expression is a compound path identifying a labelled element of a structure, as discussed above in Paths. Every prefix of a path is itself a path. That is, *every* identifier in a path is interpreted as the label of an element of a structure.

Each element of the path is assigned a type:

1. If the root element is an identifier matching an alias declared in a `from` clause of any containing query, as defined below in From clause, the root element is assigned the entity type associated with that alias.[1]
2. Otherwise, if the root element is an identifier, and if a containing query has a `from` clause declaring a root entity with (implicit or explicit) alias `this`, then the path expression is interpreted as if it were prefixed with the first element `this`, with `this` being assigned the type of the root entity.[2]
3. If the root element is a `treated_entity_path_expression`, then it is assigned the treated type, according to the usual rules defined by Treated path expressions.
4. If the root element is a map key or value expression, then it is assigned a type according to the usual rules specified by Map keys and values.
5. The identifier of each non-root element must match the label of an element of the entity or embeddable type assigned to the previous element. Then the non-root element is assigned the type of this element of the entity or embeddable type.

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

> Typically, the last element of a path expression is assigned an atomic value type. Non-terminal path elements are assigned a structure type, usually an embeddable type, if the element references an embeddable, or an entity type, if the element references an association. Jakarta EE specifications that use Jakarta Query determine whether a provider is required to support embedded attributes or associations, which require support for compound path expressions.

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

- The root element of the path expression is evaluated in the context of a given structure belonging to a result list, and:
  - if the root element is an identifier, it evaluates to the value of the labeled element of the given structure, or
  - otherwise, it is evaluated as a value expression according to the rules specified here.
- Each subsequent element of the path is evaluated in the context of the structure produced by evaluating the previous element (typically, an instance of an entity or embeddable type), and evaluates to the value of the labeled element of the result structure.

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

> An implementation of the core language is not required to support path expressions which involve multiple entities. For example, if `Book` and `Publisher` are different entity types, then the path expression `book.publisher.name` is not part of the core language.[3]

5.1.7. Identifier and version expressions

In the core language, 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.

In the extended language, the grammar rule `entity_id_or_version_function` gives the syntax of the special functions `id` and `version`, which accept a path expression whose last element is assigned an entity type, and evaluate to, respectively, the identifier or version element of the record to which the path expression evaluates when executed.

Record identifiers were defined above in Structures and records. This specification leaves the notion of a *version* undefined. Implementations of Jakarta Query are free to interpret this notion in terms of concepts defined externally to this specification. For example, Jakarta Persistence specifies the notion of a version field or property of an entity.

5.1.8. Function calls

A *function call* is the name of a function recognized by the Jakarta Query implementation, followed by a parenthesized list of argument expressions, with syntax given by:

- `function_expression` in the core language, or
- `functions_returning_strings`, `functions_returning_datetime`, and `functions_returning_numerics` in the full grammar of the extended language.

This specification defines the standard functions listed in the table below.

Functions highlighted in yellow belong to the extended language and are not required for an implementation of the core language.

Function name	Parameters	Parameter types	Type	Semantics
`abs`	1	Any numeric type	Same as argument	Evaluates to the absolute value of the numeric value to which its argument evaluates.
`sign`	1	Numeric	Integer	Evaluates to the sign (-1, 0, or 1) of the numeric value of its argument.
`mod`	2	Both integer	Integer	Evaluates to the remainder when its first integer argument is divided by its second integer argument. The behavior is undefined when either or both of the arguments are negative and depends on the data store that is used.
`sqrt`	1	Numeric	Double precision	Evaluates to the positive square root of its numeric argument.
`exp`	1	Numeric	Double precision	Evaluates to the natural exponential of its numeric argument.
`ln`	1	Numeric	Double precision	Evaluates to the natural logarithm of its numeric argument.
`power`	2	Both numeric	Double precision	Evaluates to the value produced by raising its first numeric argument to the power specified by its second numeric argument.
`ceiling`	1	Any numeric type	Same as argument	Evaluates to the smallest integral value at least as large as its argument.

Function name	Parameters	Parameter types	Type	Semantics
`floor`	1	Any numeric type	Same as argument	Evaluates to the largest integral value at least as small as its argument.
`round`	2	Any numeric type, integer	Same as first argument	Evaluates to the value produced by rounding its first numeric argument with the precision given by its second integer argument.
`length`	1	String	Integer	Evaluates to the length of string to which its argument evaluates.
`lower`	1	String	String	Evaluates to the lowercase form of the string to which its argument evaluates.
`upper`	1	String	String	Evaluates to the uppercase form of the string to which its argument evaluates.
`left`	2	String, integer	String	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.
`right`	2	String, integer	String	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.
`concat`	At least one	All strings	String	Evaluates to the concatenation of its arguments.
`substring`	2 or 3	String, integer, integer	String	Evaluates to a specified substring of the first argument. The second and third arguments specify the starting position and length of the substring. The third argument is optional. If it is not specified, the substring from the starting position to the end of the string is returned. The first character of the string is at position `1`.
`trim`	1*	String	String	Trims a specified character from its last argument. If the character to be trimmed is not specified, the space character is trimmed. The optional `trim_character` specifies the character to be trimmed. The optional `trim_specification` controls whether the character is trimmed from the start and/or end of the string. By default, the character is trimmed from both start and end.
`replace`	3	All strings	String	Evaluates to a new string formed by replacing every occurrence of the second argument string within the first argument string with the third argument string.

Function name	Parameters	Parameter types	Type	Semantics
`locate`	2 or 3	String, string, integer	Integer	Evaluates to the position at which one string occurs within a second string, optionally ignoring any occurrences that begin before a specified character position in the second string. It returns the first character position within the second string (after the specified character position, if any) at which the first string occurs, as an integer, where the first character of the second string is denoted by 1. That is, the first argument is the string to be searched for; the second argument is the string to be searched in; the optional third argument is an integer representing the character position at which the search starts (by default, 1, the first character of the second string). If the first string does not occur within the second string, 0 is returned.
`size`	1	Collection	Integer	Evaluates to the number of elements in the collection to which its argument evaluates.
`coalesce`	At least two	Any atomic type `T`	`T`	Evaluates to the value of the first argument expression which evaluates to a non-null value.
`nullif`	2	Any atomic type `T`	`T`	Evaluates to the null value if both argument expressions evaluate to the same value, or, otherwise, to the value of the first argument expression.
`extract`	2	Any date or time type	See text	See text

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

Some of these functions cannot be emulated on every datastore. When a function cannot be reasonably emulated via the native query capabilities of the database, an implementation of Jakarta Query is not required to provide the function.

On the other hand, an implementation of Jakarta Query might provide additional built-in functions, and might even allow invocation of user-defined functions.

An implementation Java must assign:

- the type `java.lang.String` to every function of type "String",
- the type `java.lang.Integer` to every function of type "Integer", and
- the type `java.lang.Double` to every function of type "Double precision".

The primitive types `double`, `float`, `long`, `int`, `short`, `byte`, wrappers for these primitive types, `BigInteger`, and `BigDecimal` are all considered "Numeric" types.

In the extended language, the `extract()` function accepts an expression assigned a date, time, or datetime type, along with an identifier — a `datetime_field` or `datetime_part` — indicating a specific part of the date, time, or datetime to extract, and evaluates to the specified part of the value to which its argument expression evaluates.

Like keywords, `datetime_field` and `datetime_part` identifiers are case-insensitive.

As mentioned above in Atomic values, an implementation of Jakarta Persistence for Java usually supports at least the date/time types `java.time.LocalDate`, `java.time.LocalTime`, and `java.time.LocalDateTime`. Such implementations are encouraged to also support:

- the `datetime_field` identifiers `YEAR`, `QUARTER`, `MONTH`, `WEEK`, `DAY`, `HOUR`, `MINUTE`, `SECOND`, and
- the `datetime_part` identifiers `DATE` and `TIME`.

If the first argument of `extract()` is a `datetime_field` identifier, the function call is assigned the type `Integer`. If the first argument of `extract()` is `DATE`, the function call is assigned the type `LocalDate`. If the first argument of `extract()` is `TIME`, the function call is assigned the type `LocalTime`.

The `datetime_field` or `datetime_part` must be compatible with the type of the second argument expression. For example, `extract(day from local date)` is well-typed; `extract(year from local time)` is not.

Jakarta Persistence requires support for the `function()` function, with syntax given by `function_invocation`, allowing invocation of a native or user-defined database function from a query written in JPQL. On the other hand, an implementation of Jakarta Query might simply allow direct invocation of such functions — without the requirement to use the `function()` syntax — as an extension to the functionality required by this specification. This specification does not, therefore, require support for `function()`, not even in an implementation of the extended language.

TODO Should we simply deprecate it? Remove it?

5.1.9. Map keys and values

In the extended language, a *map key expression*, *map value expression*, or *map entry expression* is an application of the special `key()` function, special `value()` function, or special `entry()` function, respectively.

The `entry()` function may only occur in the select clause, as specified by the full grammar of the extended language.

The argument of the `key()`, `value()`, or entry() function must be an alias of a collection join of a map, as defined in Join clauses.

TODO This is very restrictive. Should we allow application of `key()` or `value()` to a path expression?

- A map key expression is assigned the type of the key of the map. A map key expression evaluates to the key of the map entry to which its argument expression evaluates.
- A map value expression is assigned the type of the value of the map. A map value expression evaluates to the value to which its argument expression evaluates.

Application of the `value()` value function is always optional.

An implementation for Java must assign the type `java.util.Map.Entry<K,V>` to a map entry expression, where `K` is the type of the key of the map, and `V` is the type of the value of the map. The map entry expression evaluates to an instance of this type packaging the value to which its argument expression evaluates and the key of the map entry of this value.

An implementation of Jakarta Query which targets clients written in a language other than Java is not required to provide support for the `entry()` function.

5.1.10. Types and typecasts

The extended language provides three special functions for working with subtype polymorphism.

5.1.10.1. Treated path expressions

A *treated path expression* is an invocation of the special `treat()` function, with syntax given by `treated_entity_path_expression` and `treated_joinable_path_expression`. The `treat()` function accepts:

1. a path expression whose last element is assigned an entity type, and
2. the name of an entity type — called the *treated type* — which must be a subtype of the entity type assigned to the path expression.

A treated path expression is assigned the treated type.

When a treated path expression is executed, the record produced by evaluating the path expression is compared to the treated type.

- If the record is an instance of the treated type, the treated path expression evaluates to the record.
- Otherwise, if the record is not an instance of the treated type, the treated path expression evaluates to the null value.

5.1.10.2. Coercion expressions

A *coercion expression* is an invocation of the special `cast()` function, with syntax given by the last alternatives of `functions_returning_numerics` and `functions_returning_strings`. The `cast()` function accepts:

1. an expression assigned an atomic type, and
2. the name of an atomic type.

A coerced expression is assigned the named atomic type.

When a coercion expression is executed, the atomic value produced by evaluating the path expression is coerced to the named atomic type.

This specification places no specific requirements on the types which are allowed as arguments of the `cast()` function, nor on the behavior of coercion between types. As suggested by the grammar for `functions_returning_numerics` and `functions_returning_strings`, implementations of the extended language are strongly encouraged to support at least:

- coercion from string to any numeric type, and
- coercion from any atomic type to string.

However, this part of the grammar should be read as indicative of what should be supported in implementations for Java, and, more specifically, what is required for an implementation of Jakarta Persistence.

The capabilities of the `cast()` function vary between client programming languages and between databases.

5.1.10.3. Entity type expressions

An *entity type expression* is an invocation of the special `type()` function, with syntax given by `type_discriminator`. The `type()` function accepts an expression assigned an entity type, and, when executed, evaluates to a value which represents the type of the record to which the argument expression evaluates.

An implementation for Java assigns the type `java.lang.Class<? extends E>` where `E` is the Java class mapping the entity type assigned to the argument expression, and an entity type expression evaluates to an instance of `Class` representing the Java class which maps the entity type of the record to which the argument expression evaluates.

5.1.11. Aggregate functions calls

An *aggregate function call* may only occur in the `select` or `having` clause of a query involving aggregation. Such a clause operates on a list of nested result lists, as specified above in Projection and aggregation and Restriction and aggregation. An aggregate function call is evaluated in the context of such a nested list.

This specification defines the standard aggregate functions listed in the table below.

Functions highlighted in yellow belong to the extended language and are not required for an implementation of the core language.

Function name	Parameters	Parameter types	Type	Semantics
`count`	1	Any type	Long integer	The number of nested list elements for which the argument expression evaluates to a non-null value
`min`	1	Any ordered type `O`	`O`	The smallest non-null value of the argument expression over all nested list elements
`max`	1	Any ordered type `O`	`O`	The largest non-null value of the argument expression over all nested list elements
`sum`	1	Any numeric type `N`	`N`	The sum of non-null values of the argument expression over all nested list elements
`avg`	1	Any numeric type `N`	`N`	The average (arithmetic mean) of non-null values of the argument expression over all nested list elements

In the core language, the only allowed aggregate function call is the expression `count(this)`, as specified below in Select clause.

In the extended language, the syntax for aggregate functions is given by `aggregate_expression`. An aggregate function invocation may specify the keyword `distinct`, in which case duplicate elimination is applied to the list of values produced by evaluating the argument expression over all elements of the nested list before counting or summing the values.[4]

5.1.12. Case expressions

In the extended language, a *case expression* has syntax given by `general_case_expression` or `simple_case_expression`.

A `general_case_expression` has:

1. a list of one or more `when_clause`s, each of which has a `conditional_expression` paired with a result `scalar_expression`, and,
2. optionally, a default `scalar_expression`.

When a general case expression is executed, each `conditional_expression` is evaluated, in order, until one is satisfied.

- If some `conditional_expression` is satisfied, then its result `scalar_expression` is evaluated.
- Otherwise, if no `conditional_expression` is satisfied, the default `scalar_expression`, if any, is evaluated.

The whole case expression evaluates to the value produced by the result or default `scalar_expression` which was evaluated. If there is no `else`, and no `scalar_expression` was evaluated, the whole case expression evaluates to the null value.

A `simple_case_expression` has:

1. a `case_operand`, which must be an atomic-valued path expression or an entity type expression,
2. a list of one or more `simple_when_clause`s, each of which has a tested `scalar_expression` paired with a result `scalar_expression`, and,
3. optionally, a default `scalar_expression`.

When a simple case expression is executed, the `case_operand` is evaluated, and then each tested `scalar_expression` is evaluated, in order, until one produces a value identical to the value of the `case_operand`.

- If some tested `scalar_expression` evaluates to the value of the `case_operand`, then its result `scalar_expression` is evaluated.
- Otherwise, if no `conditional_expression` is satisfied, the default `scalar_expression`, if any, is evaluated.

The whole case expression evaluates to the value produced by the result or default `scalar_expression` which was evaluated. If there is no `else`, and no `scalar_expression` was evaluated, the whole case expression evaluates to the null value.

5.1.13. Operator expressions

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

- The operands of `+`, `-`, `*`, and `/` must be expressions assigned an atomic numeric type. The operators have their usual interpretation in terms of integer or floating point arithmetic, subject to the rules of numeric promotion.
- The operands of `||` must be expressions assigned an atomic type representing character strings. An `||` operator expression evaluates to the concatenation of the character strings obtained by evaluating its operands.

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 precise behavior of numeric operators is outside the scope of this specification and varies according to the database and client programming language.

In an implementation for Java:

- 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 as they have in Java, and operator expressions involving these operators are assigned the types they would be assigned in Java. As an exception, when the operands of `/` are both integers, an implementation of Jakarta Query 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.[5]

When a Jakarta Query implementation targets a non-relational database, support for arithmetic operators or support for the use of parentheses to control operator precedence might vary from what is described above. This specification does not require support for arithmetic operators or grouping parentheses if the underlying datastore does not provide these features among its native querying capabilities.

5.1.14. Subquery expressions

In the extended language, a `subquery` may occur as a `scalar_expression` in many places where an expression of atomic type is legal. Such a subquery must produce a result list with exactly one element when executed. When the `scalar_expression` is evaluated, the subquery is executed, and the whole expression evaluates to the single element of the result list of the subquery.

5.1.15. 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. Numeric type promotion is defined by the following rules:

- 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.

5.2. 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.

> The Jakarta Persistence specification defines the result of a conditional expression in terms of ternary logic. This specification does not specify that a conditional expression evaluates to a 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,
- quantified conditional expressions, and
- logical operator expressions.

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

> When a Jakarta Query implementation targets a non-relational database, support for conditional expression might depart from what is described below. This specification does not require support for use of any sort of conditional expression if the underlying datastore does not provide a functionally equivalent capability among its native querying capabilities.[6] For example, in a non-relational database which only allows lookup by key, the only sort of conditional expression which is allowed is an equality operator expression involving the entity identifier.[7]

5.2.1. 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.

5.2.2. In expressions

In the core language, an `in` expression, with syntax given by `in_expression`, must have:

- a path expression as the leftmost operand, and
- a parenthesized list of one or more `in_items`, each of which must be a literal, enum literal, or parameter.

All operands must have the same type.

TODO We should relax the restrictions on the left operand.

In the extended language, the leftmost operand must be a path expression assigned an atomic type or an entity type expression, and there must be exactly one of the following elements:

- a parenthesized list of one or more `in_item`s, each of which must be a literal, an enum literal, a literal entity type, `true`, `false`, or a parameter, and each having the same assigned type as the leftmost operand,
- a parenthesized subquery with a `simple_select_expression` expression with the same assigned type as the leftmost operand, or
- an unparenthesized parameter.[8]

If the condition has a list of expressions, it 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.

If the condition has a subquery, it is satisfied when its leftmost operand evaluates to a non-null value, and:

- the `not` keyword is missing, and the value produced by evaluating the left operand occurs in the result list of the subquery, or
- the `not` keyword occurs, and the value produced by evaluating the left operand does not occur in the result list of the subquery.

If the condition has an unparenthesized parameter, it is satisfied when its leftmost operand evaluates to a non-null value, and:

- the `not` keyword is missing, and the value produced by evaluating the left operand is a member of the value assigned to the parameter, or
- the `not` keyword occurs, and the value produced by evaluating the left operand is not a member of the value assigned to the parameter.

5.2.3. 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 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.

5.2.4. Like expressions

A `like` expression has syntax given by `like_expression`.

- Its left operand must be an expression assigned some atomic type representing character strings.
- Its right operand must be a pattern given as a literal string, with syntax `literal_pattern`, or, in the extended language, a parameter.
- Optionally, its right operand may specify an `escape_character` as a literal single character.

If an implementation of Jakarta Query targets clients written in Java, it must allow expressions assigned the type `java.lang.String` as the left operand of a `like` expression.

The 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.

Matching is lexicographic. Within the pattern, the character bigram `''` is interpreted as a literal `'`, and the characters `_` and `%` are interpreted as wildcards:

- `_` matches any single character, and
- `%` matches any contiguous sequence of (zero or more) characters.

Any other character occurring in the pattern is interpreted literally and matches only itself.

An escape character may be used to suppress the interpretation as a wildcard of a particular `_` or `%` character in the pattern.

That is, if the escape character is `c`, then the character bigram `c_` is interpreted literally as the character `_`, the bigram `c%` as the character `%`, and the bigram `cc` as the character `c`.

5.2.5. Equality and inequality operators

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

The operands of an equality or inequality operator must have the same type, and it must be an atomic type considered *comparable* by the implementation of Jakarta Query. Any type with a natural order must be treated as a comparable type, and the equality and inequality operators must be interpreted in a way which is consistent with the natural order.

For non-null values, the following table defines an interpretation consistent with the natural order:

Operator	**Interpretation in terms of natural order**
`x < y`	Satisfied if `x` occurs before `y` in the natural order
`x > y`	Satisfied if `x` occurs after `y` in the natural order
`x ⇐ y`	Satisfied if `x>y` is not satisfied
`x >= y`	Satisfied if `x<y` is not satisfied
`x = y`	Satisfied if neither `x>y` nor `x<y` is satisfied
`x <> y`	Satisfied if either `x>y` or `x<y` is satisfied

When exactly one of the operands of an equality or inequality operator evaluates to a null value, the conditional expression is not satisfied.

When both operands of an equality or inequality operator evaluate to null, the behavior is not defined by this specification. As mentioned above in Atomic values, the semantics of comparisons involving two null values depends on the underlying database — typically, on whether the database uses binary or ternary logic.

In an implementation of Jakarta Query which targets clients written in Java, every primitive or primitive wrapper type is considered comparable, along with `java.lang.String`, `java.math.BigInteger`, `java.math.BigDecimal`, and types representing dates and datetimes.

- 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 usually evaluated according to the native semantics of the database.

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. [9]

In the extended language, the right operand of an equality or inequality operator may be a subquery, as discussed in the next section.

5.2.6. Quantified conditional expressions

In the extended language, there are two kinds of conditional expression involving quantification:

- *existence expressions*, with syntax given by `exists_expression`, and
- *quantified comparison expressions*, in which the right operand of an equality or inequality operator is an `all_or_any_expression`.

In either case, a `subquery` occurs as the argument of one of the following special quantifier functions: `exists`, `all`, `any`, or `some`. When a quantifier is evaluated, the subquery is executed, producing a result list containing values which are instances of some atomic or entity type. For a quantified comparison expression, the type of these values must be the same as the type of the left operand of the equality or inequality operator.

An existence expression is satisfied if and only if:

- the `not` keyword is missing, and the result list of the subquery is nonempty, or
- the `not` keyword is present, and the result list of the subquery is empty.

A quantified comparison expression is satisfied if and only if:

- the quantifier is `any` or `some` and the result list of the subquery contains at least one value which would make the equality or inequality expression evaluate to true if it occurred as the right operand in place of the subquery, or
- the quantifier is `all` and every value belonging to the result list of the subquery would make the equality or inequality expression evaluate to true if it occurred as the right operand in place of the subquery.

5.3. Natural order

Every atomic value 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 any numeric type, or for any date or datetime type, there is a *natural order* for non-null values completely determined by the semantics of the type. Jakarta Query implementations must sort these types according to their natural order:

- smaller numeric values come before larger numeric values, and
- earlier dates or datetimes come before later dates or datetimes.

For an implementation of Jakarta Query which targets clients written in Java, this natural order agrees with the order defined by the corresponding Java type from `java.lang`, `java.math`, or `java.time`.

Furthermore, Jakarta Query implementations must sort boolean values 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.

For clients written in Java, note that:

- Textual data is represented in Java 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 Jakarta Query is not required to emulate Java collation.
- Binary data is represented in Java 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 Jakarta Query to respect it.
- This specification does not define an order for the sorting of Java `enum` values, which is provider-dependent. An implementation of Jakarta Query 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 completely implementation-dependent. Applications must not assume that the order agrees with the `compareTo()` method of `java.util.UUID`.

The natural order does not determine the precedence of null values, and so this specification leaves their precedence undefined except when `nulls first` or `nulls last` is explicitly specified for an item of the order by clause, as defined by Order clause.

5.4. 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 Jakarta Query 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 a Jakarta Query implementation targets a non-relational database, support for logical operators might vary from what is described above. This specification does not require support for logical operators if the underlying datastore does not provide a functionally equivalent way to restrict query

> results among its native querying capabilities.

[1] This is the same type as the type of a structure with that label belonging to each element of the result list.

[2] This is the same type as the type of a structure with label `this` belonging to each element of the result list.

[3] Path expressions involving multiple entities are called *implicit joins* by the Jakarta Persistence specification.

[4] Use of `min(distinct …)` or `max(distinct …)` is allowed but redundant.

[5] As earlier noted, this specification never mandates an inefficient implementation of operations which are implemented by the database itself.

[6] This is not an open-ended invitation for implementations backed by non-relational databases to ignore the requirements of this specification in cases where there *is* a reasonable way to emulate the behavior of a conditional expression.

[7] In Jakarta NoSQL, the key attribute is identified by the annotation `jakarta.nosql.Id`.

[8] Jakarta Persistence treats this parameter as a *collection-valued input parameter*.

[9] A notable special case is `java.math.BigDecimal`, whose implementation of `equals()` is inconsistent with its implementation of `compareTo()`. For `BigDecimal`, the `=` operator in Jakarta Data should be implemented for consistency with `compareTo()`.

Chapter 6. Statements and clauses

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

6.1. Clauses

As discussed in Basic operations, 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. `group`,
4. `having`,
5. `select` or `set`,
6. `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.

6.1.1. From clause

In the core language, the `from` clause, with syntax given by `from_clause`, specifies an *entity name* identifying the root entity of the query, as defined in The root entity. This entity is always assigned the implicit alias `this`.

In the core language, 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 Entities and embeddable types.

The core language does not have joins, and so the `from` clause is always trivial.

In the extended language, the `from` clause might enumerate multiple steps of the basic operation pipeline. The `from_clause` of the full grammar allows:

1. A root entity, as defined in The root entity, which may either:
   - explicitly specify its alias according to the syntax given by `range_variable_declaration`, or
   - let the alias default to `this`, by using the syntax given by `this_implicit_variable`.
2. Optionally, a list of additional `range_variable_declaration`s interpreted as joins to named entities.
3. Optionally, one or more join clauses, as defined in Joins, with syntax given by `join`, each interpreted as a join to a named entity or as a join to a nested structure or collection, which may each optionally also contain a restriction with syntax given by `join_condition`.

When the root entity is declared using the rule `this_implicit_variable`, the query must not have joins. A query with joins must use an `identification_variable_declarations` list to specify the root entity and joins.

When a `from` clause has multiple elements in the `identification_variable_declarations` list, the root entity is always the first `identification_variable_declaration` to occur in the list. Each subsequent `identification_variable_declaration` is considered a join to a named entity. Join clauses are discussed below.

An `identification_variable_declaration` declaring a root entity has a `range_variable_declaration` with an `entity_name` specifying the name of the root entity, and an `identification_variable` specifying its alias (label).

Since a `from` clause may introduce multiple distinct labels, paths in the extended language may begin with the label of an entity. Alternatively, a path which does not begin with the label of an entity is interpreted as beginning with the implicit label `this`.

The `from` clause is optional in `select` statements when a mechanism from an external specification is used to identify the queried entity. For example:

1. a result type might be supplied via an API invocation (for example, via `createQuery()` in Jakarta Persistence), or
2. it might be determined by static analysis (for example, in Jakarta Data, by the return type of a repository method or by the primary entity type of the repository).

For example, this query provided to the Jakarta Persistence `createQuery()` method with result type of `Book.class` or supplied to a Jakarta Data repository method that returns `List<Book>`:

```
where title like :title
```

is equivalent to:

```
from Book where title like :title
```

6.1.2. Join clauses

In the extended language, an `identification_variable_declaration` might include a list of:

- joins, with syntax given by `join` and semantics specified by Joins, and
- fetch joins, with syntax given by `fetch_join` and semantics specified by Fetch joins.

Each `join` is in turn either a `range_join`, with semantics given by Joins to named entities or a `path_join`, with semantics given by Joins to nested entities or collections. Such joins may specify a restriction.

In addition, each `identification_variable_declaration` apart from the first — which specifies the root entity — is itself considered a join. Every such `identification_variable_declaration` is interpreted as a regular (inner) join and never specifies a restriction.

Every join has a *target*, either a named entity type belonging to the database, or a path to a nested structure or collection in the result list of the previous operation in the pipeline:

- A `range_join` or `identification_variable_declaration` has a `range_variable_declaration` with an `entity_name` specifying the name of the joined entity.
- A `path_join` has a `join_association_expression` specifying the joined nested structure or collection.

In either case, a `join_condition` may occur, specifying a restriction which is applied to the result list of the join.

A `range_join` or `identification_variable_declaration` has a `range_variable_declaration` with an `identification_variable`. Similarly, a `path_join` has a `identification_variable`. The `identification_variable` specifies the alias (label) introduced by the join.

Every `join` or `fetch_join` begins with a `join_spec`:

- if the keyword `left` occurs in the `join_spec`, then the join is interpreted as a left (outer) join, as specified by Left joins, or
- otherwise, if the keyword `left` is missing, the join is interpreted as a regular (inner) join.

The joins belonging to a query are evaluated in the order in which they occur in the query.

6.1.3. Where clause

The `where` clause, with syntax given by `where_clause`, specifies a restriction on a result list, as specified by Restriction.

The `where` clause specifies a `conditional_expression`, interpreted as the logical predicate of the restriction.

A restriction specified using `where` is applied after every join and before any aggregation or projection.

6.1.4. Group clause

The `group` clause in the extended language, with syntax given by `groupby_clause`, specifies a list of grouping items, which are collectively interpreted as an aggregation, as specified in Aggregation.

According to the rule `groupby_item`, each item must be either:

- an atomic-valued path expression, with syntax given by `atomic_valued_path_expression`, or
- an entity-valued path expression, with syntax given by `entity_valued_path_expression`.

Each such path expression is interpreted as a value expression producing an element of the grouping tuple of the aggregation.

An implementation of Jakarta Query might treat the entity-valued path expression as an expression referring to the identifier of the entity.

6.1.5. Having clause

The `having` clause in the extended language, with syntax given by `having_clause`, specifies a restriction on the result list of an aggregation, as specified by Restriction.

The `having` clause specifies a `conditional_expression`, interpreted as the logical predicate of the restriction.

The `having` clause must follow a `group` clause. A query with no group clause may not have a having clause. A restriction specified using `having` is applied after aggregation and before any projection.

6.1.6. Select clause

In the core language, the `select` clause, with syntax given by `select_clause`, specifies one or more path expressions with syntax `simple_path_expression`, which are collectively interpreted as a projection with automatically assigned aliases, as specified in Projection.

Alternatively, the `select` clause may contain either:

- a single `count(this)` aggregate function invocation, or
- a single identifier expression.

A query beginning with `select count(this)` performs aggregation (without grouping), as specified by Projection and aggregation and always returns a result list of unit length.

When a Jakarta Query implementation targets a non-relational database, support for projection might vary from what is described above. In particular, this specification does not require support for `count(this)` if the underlying datastore only provides for key-based lookups.

In the extended language, the `select` clause is much more flexible. Its syntax, given by `select_clause`, permits:

- one or more expressions, with optional labels, as given by select_item, where select_expression is the expression, and result_variable is the label, and
- optionally, the distinct keyword.

The list of expressions is interpreted as a projection, as specified by Projection, and, if at least one of the expressions is an aggregate function, also aggregation, as specified in Projection and aggregation.

If the distinct keyword occurs, then projection is followed by duplicate elimination, as specified by Duplicate elimination.

6.1.7. 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 satisfying 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.

6.1.8. 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 expressions used to sort the records which satisfy the restriction imposed by the where clause, as specified in Ordering.

An item of the order by clause, with syntax given by orderby_item has:

- an orderby_expression, which must be an expression assigned an atomic type, and
- optionally, the keyword asc or desc,
- optionally, in the extended language, nulls first or nulls last.

In the core language, an orderby_expression must be a path expression or identifier expression. In the extended language, an orderby_expression may be almost any sort of atomic-valued expression (but not a parameter), as specified by the full grammar of the extended language.

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

The keyword asc or desc specifies that the values of a given expression should be sorted in ascending or descending order respectively; when neither is specified, ascending order is the default:

- *ascending order* is determined by the natural order of the expression type, as specified by Natural order, and
- *descending order* is the reverse of the natural order.

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.

In the extended language, nulls first or nulls last specifies whether null values should be considered, respectively, smaller or larger than non-null values. This specification does not define how null values are ordered with respect to non-null values when the sorting of null values is not explicitly specified. 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, the order of the query results is not defined by this specification and might not be deterministic.

An implementation of Jakarta Query might provide some other facility to specify sorting criteria for the results of a given query. For example, Jakarta Query allows an object carrying sorting criteria to be passed as an argument to a repository method.

When a Jakarta Query implementation targets a non-relational database, support for sorting query results might vary from what is described above. This specification does not require support for sorting a query result list according to a given ordering if the underlying datastore does not provide a way to sort query results according to that ordering via its native querying capabilities.

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.

6.2. Union, intersect, and except

The semantics of union, intersection, and complementation of query result sets is specified by Union, intersection, and complement above.

These operations are part of the extended language, and so support for union, intersection, and complementation is not required for an implementation of Jakarta Query core.

Each of these operations is treated as an infix binary operator acting on query result lists of identical shape (type) and producing a new query result set of the same shape as the operands.

6.2.1. Union and complement

The syntax of `union`, `union all`, `except`, and `except all` is given by the rule `union` of the full grammar. The operands must produce result lists of the same type. The type of the union or complement expression is the same as the type of its operands.

6.2.2. Intersection

The syntax of `intersect` and `intersect all` is given by the rule `intersection` of the full grammar. The operands must produce result lists of the same type. The type of the intersection expression is the same as the type of its operands.

6.3. 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.

6.3.1. Select statements

A select statement, with syntax given by select_statement, returns data to the client.

In the core language, a select statement may contain one of more of the following clauses:

- select
- from
- where
- order.

A select statement is a pipeline as defined in Basic operations. The result list of the whole select statement is the same as the result list of the last operation in the pipeline.

In the extended language, a select statement may contain, in addition, a group and/or a having clause, with syntax specified by select_query. Each select_query is a pipeline as defined in Basic operations. A select_statement in the full grammar may contain union, intersection, and complement operations, as defined in Union, intersect, and except.

- If the select_statement is a union or intersection, the result list of the whole select statement is the same as the result list of the union or intersection.
- Otherwise, if the select_statement is just a select_query, result list of the whole select statement is the same as the result list of the last operation in the pipeline.

The select clause is optional in a select statement. A query with a missing select clause is interpreted as if it had the following single-item select clause:

```
select this
```

where this is the implicit alias.

6.3.2. Update statements

An update statement, with syntax given by update_statement, updates each record belonging to the entity type named in the update clause which satisfies the restriction imposed by the where clause, if any.

An update statement does not have a well-defined result, but implementations typically return the number of matching records to the client. Such functionality falls outside the scope of this specification.

A Jakarta Query implementation which targets a non-relational database might not support conditional updates. This specification does not require support for conditional updates if the underlying datastore does not provide a functionally equivalent capability.[1]

6.3.3. Delete statements

A delete statement, with syntax given by delete_statement, deletes each record belonging to the entity type named in the update clause which satisfies the restriction imposed by the where clause, if any.

A delete statement does not have a well-defined result, but implementations typically return the number of deleted records to the client. Such functionality falls outside the scope of this specification.

A Jakarta Query implementation which targets a non-relational database might not support conditional deletes. This specification does not require support for conditional deletes if the underlying datastore does not provide a functionally equivalent capability.

[1] Alternatively, in a non-relational database with *append-only semantics* — common in time-series and wide-column databases — conditional updates might be provided, but the update operation might behave more like an insert operation, with repeated updates to the same record not overwriting previous values.

Chapter 7. Syntax

The following grammars define the syntax of the Jakarta Query language, via ANTLR4-style BNF.

7.1. Core language grammar

The following grammar is the grammar for the core language.

```
grammar JQLCore;

statement : select_statement | update_statement | delete_statement;

select_statement : select_clause? from_clause? where_clause? orderby_clause?;
update_statement : update_clause set_clause where_clause?;
delete_statement : delete_clause where_clause?;

from_clause : 'FROM' this_implicit_variable;
this_implicit_variable : entity_name;

where_clause : 'WHERE' conditional_expression;

update_clause : 'UPDATE' entity_name;
set_clause : 'SET' update_item (',' update_item)*;
update_item : simple_path_expression '=' new_value;
new_value
    : scalar_expression
    | 'NULL'
    ;

delete_clause : 'DELETE' 'FROM' entity_name;

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

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

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 comparison_operator scalar_expression;
between_expression : scalar_expression 'NOT'? 'BETWEEN' scalar_expression 'AND' scalar_expression;
like_expression : scalar_expression 'NOT'? 'LIKE' escaped_pattern;
```

```
comparison_operator
    : '='
    | '>'
    | '>='
    | '<'
    | '<='
    | '<>'
    ;

escaped_pattern
    : literal_pattern
      ('ESCAPE' escape_character)?
    ;

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

null_comparison_expression : simple_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
    | simple_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
    : special_boolean_expression
    | special_datetime_expression
    ;

special_boolean_expression
    : 'TRUE'
    | 'FALSE'
    ;

special_datetime_expression
    : 'LOCAL' 'DATE'
    | 'LOCAL' 'TIME'
    | 'LOCAL' 'DATETIME'
    ;

simple_path_expression : IDENTIFIER ('.' IDENTIFIER)*;

entity_name : IDENTIFIER; // no ambiguity

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

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

literal : string_literal | numeric_literal;

numeric_literal : INTEGER | DOUBLE;

string_literal : STRING;

literal_pattern : STRING;

escape_character : CHARACTER;
```

7.2. Full language grammar

This much longer grammar is the grammar for the full language.

```
grammar JQLFull;

statement
    : select_statement
    | update_statement
    | delete_statement
    ;

select_statement
    : union
    ;

union
    : intersection
    | union
      ('UNION' 'ALL'? | 'EXCEPT' 'ALL'?)
      intersection
    ;

intersection
    : query_expression
    | intersection
      'INTERSECT' 'ALL'?
      query_expression
    ;

query_expression
    : select_query
    | '(' union ')'
    ;

select_query
    : select_clause?
      from_clause
      where_clause?
      groupby_clause?
      having_clause?
      orderby_clause?
    ;
```

```
update_statement
    : update_clause set_clause where_clause?
    ;

delete_statement
    : delete_clause where_clause?
    ;

from_clause
    : 'FROM' (this_implicit_variable | identification_variable_declarations)
    ;

this_implicit_variable
    : entity_name
    ;

identification_variable_declarations
    : identification_variable_declaration
      (',' identification_variable_declaration)*
    ;

identification_variable_declaration
    : range_variable_declaration
      (join | fetch_join)*
    ;

range_variable_declaration
    : entity_name
      'AS'? identification_variable
    ;

join
    : range_join
    | path_join
    ;

range_join
    : join_spec range_variable_declaration
      join_condition?
    ;

path_join
    : join_spec join_association_expression
      'AS'? identification_variable
      join_condition?
    ;

fetch_join
    : join_spec 'FETCH' join_association_expression
    ;

join_spec
    : ('INNER' | 'LEFT' 'OUTER'?)?
      'JOIN'
    ;

join_condition
    : 'ON' conditional_expression
    ;

join_association_expression
    : joinable_path_expression
    | treated_joinable_path_expression
    ;

treated_joinable_path_expression
    : 'TREAT' '(' joinable_path_expression 'AS' subtype ')'
    ;

joinable_path_expression
    // Note that unlike in derived_path_expression,
    // JPQL does not allow use of TREAT() here
    // TODO: Should we allow KEY(), VALUE() here?
    : (identification_variable '.')?
      (structure_field '.')*
      (entity_field | collection_field)
    ;

map_entry_identification_variable
    : 'ENTRY' '(' identification_variable ')'
    ;

map_keyvalue_identification_variable
    : 'KEY' '(' identification_variable ')'
    | 'VALUE' '(' identification_variable ')'
    ;

single_valued_path_expression
    : atomic_valued_path_expression
    | embeddable_valued_path_expression
    | entity_valued_path_expression
    | map_entry_identification_variable
    ;

atomic_valued_path_expression
    : (root_entity_expression '.')?
      (structure_field '.')*
      atomic_field
    | map_keyvalue_identification_variable
    ;

embeddable_valued_path_expression
    : (root_entity_expression '.')?
      (structure_field '.')*
      embedded_field
    | map_keyvalue_identification_variable
    ;

entity_valued_path_expression
    : (root_entity_expression '.')?
      (structure_field '.')*
      entity_field
    | identification_variable
    | map_keyvalue_identification_variable
    ;

collection_valued_path_expression
    : (root_entity_expression '.')?
      (structure_field '.')*
      collection_field
    ;

root_entity_expression
    : identification_variable
    | map_keyvalue_identification_variable
    | treated_entity_path_expression
    ;

treated_entity_path_expression
    : 'TREAT' '(' entity_valued_path_expression 'AS' subtype ')'
    ;

update_clause
    // TODO: Could be:
```

```
    //       'UPDATE' (this_implicit_variable | range_variable_declaration)
    : 'UPDATE' entity_name
      ('AS'? identification_variable)?
    ;

set_clause
    : 'SET' update_item (',' update_item)*
    ;

update_item
    : updatable_path_expression '=' new_value
    ;

updatable_path_expression
    // must resolve to entity or atomic type (cannot have implicit joins)
    : (identification_variable '.')?
      (embedded_field '.')*
      (atomic_field | entity_field)
    ;

new_value
    : scalar_expression
    | simple_entity_expression
    | 'NULL';

simple_entity_expression
    : identification_variable
    | single_valued_input_parameter
    ;

delete_clause
    : 'DELETE' 'FROM' entity_name
      ('AS'? identification_variable)?
    ;

select_clause
    : 'SELECT' 'DISTINCT'?
      select_item (',' select_item)*
    ;

select_item
    : select_expression
      ('AS'? result_variable)?
    ;

select_expression
    : single_valued_path_expression  // embeddables are allowed
    | scalar_expression
    | aggregate_expression
    | constructor_expression
    ;

constructor_expression
    : 'NEW' constructor_name
      '(' constructor_item (',' constructor_item)* ')'
    ;

constructor_item
    : single_valued_path_expression  // embeddables are allowed
    | scalar_expression
    | aggregate_expression
    ;

aggregate_expression
    : ('AVG' | 'MAX' | 'MIN' | 'SUM') '(' 'DISTINCT'? atomic_valued_path_expression ')'
    | 'COUNT' '(' 'DISTINCT'? (atomic_valued_path_expression | entity_valued_path_expression) ')'
    | function_invocation;
```

```
where_clause
    : 'WHERE' conditional_expression
    ;

groupby_clause
    : 'GROUP' 'BY'
      groupby_item (',' groupby_item)*
    ;

groupby_item
    : atomic_valued_path_expression
    | entity_valued_path_expression
    ;

having_clause
    : 'HAVING' conditional_expression
    ;

orderby_clause
    : 'ORDER' 'BY'
      orderby_item (',' orderby_item)*
    ;

orderby_item
    : orderby_expression
      ('ASC' | 'DESC')?
      ('NULLS' ('FIRST' | 'LAST'))?
    ;

orderby_expression
    : atomic_valued_path_expression
    | result_variable
    | scalar_expression
    ;

subquery_expression
    : '(' subquery ')'
    ;

subquery
    : simple_select_clause
      subquery_from_clause
      where_clause?
      groupby_clause?
      having_clause?
    ;

subquery_from_clause
    : 'FROM' subselect_identification_variable_declaration
      (',' subselect_identification_variable_declaration)*
    ;

subselect_identification_variable_declaration
    : identification_variable_declaration
    | derived_path_expression 'AS'? identification_variable join*
    ;

derived_path_expression
    // TODO: Is support for TREAT() here really a requirement?
    //       We don't allow it in joinable_path_expression
    // TODO: Should we allow KEY(), VALUE() here?
    : ((identification_variable | treated_entity_path_expression) '.')?
      (structure_field '.')*
      (entity_field | collection_field)
    ;
```

```
simple_select_clause
    : 'SELECT' 'DISTINCT'? simple_select_expression
    ;

simple_select_expression
    : atomic_valued_path_expression
    | entity_valued_path_expression
    | scalar_expression
    | aggregate_expression
    ;

scalar_expression
    : arithmetic_expression
    | string_expression
    | enum_expression
    | datetime_expression
    | boolean_expression
    | case_expression
    | entity_type_expression
    | entity_id_or_version_function
    ;

conditional_expression
    : conditional_term
    | conditional_expression 'OR' conditional_term
    ;

conditional_term
    : conditional_factor
    | conditional_term 'AND' conditional_factor
    ;

conditional_factor
    : 'NOT'? conditional_primary
    ;

conditional_primary
    : simple_conditional_expression
    | '(' conditional_expression ')'
    ;

simple_conditional_expression
    : comparison_expression
    | between_expression
    | in_expression
    | like_expression
    | null_comparison_expression
    | empty_collection_comparison_expression
    | collection_member_of_expression
    | exists_expression
    ;

between_expression
    : arithmetic_expression 'NOT'? 'BETWEEN' arithmetic_expression 'AND' arithmetic_expression
    | string_expression 'NOT'? 'BETWEEN' string_expression 'AND' string_expression
    | datetime_expression 'NOT'? 'BETWEEN' datetime_expression 'AND' datetime_expression
    ;

in_expression
    : (atomic_valued_path_expression | type_discriminator)  // TODO: Much too restrictive
      'NOT'? 'IN'
      (in_item_list | subquery_expression | collection_valued_input_parameter)
    ;

in_item_list
    : '(' in_item (',' in_item)* ')'
    ;
```

```
in_item
    : literal
    | enum_literal
    | entity_type_literal
    | special_boolean_expression
    | single_valued_input_parameter
    ;

like_expression
    : string_expression
      'NOT'? 'LIKE' escaped_pattern
    ;

escaped_pattern
    : pattern_value
      ('ESCAPE' escape_character)?
    ;

pattern_value
    : literal_pattern
    | single_valued_input_parameter
    ;

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

nullable_expression
    : atomic_valued_path_expression
    | entity_valued_path_expression
    | single_valued_input_parameter
    ;

empty_collection_comparison_expression
    : collection_valued_path_expression
      'IS' 'NOT'? 'EMPTY'
    ;

collection_member_of_expression
    : collection_member_element_expression
      'NOT'? 'MEMBER' 'OF'?
      collection_valued_path_expression
    ;

collection_member_element_expression
    : entity_valued_path_expression
    | atomic_valued_path_expression
    | single_valued_input_parameter
    | literal
    | enum_literal
    | special_boolean_expression
    ;

exists_expression
    : 'NOT'? 'EXISTS'
      subquery_expression
    ;

all_or_any_expression
    : ('ALL' | 'ANY' | 'SOME')
      subquery_expression
    ;

comparison_expression
    : string_expression comparison_operator (string_expression | all_or_any_expression)
    | boolean_expression equality_operator (boolean_expression | all_or_any_expression)
    | enum_expression equality_operator (enum_expression | all_or_any_expression)
    | datetime_expression comparison_operator (datetime_expression | all_or_any_expression)
    | entity_expression equality_operator (entity_expression | all_or_any_expression)
    | arithmetic_expression comparison_operator (arithmetic_expression | all_or_any_expression)
    | entity_id_or_version_function equality_operator single_valued_input_parameter
    | entity_type_expression equality_operator entity_type_expression
    ;

equality_operator
    : '='
    | '<>'
    ;

comparison_operator
    : '='
    | '>'
    | '>='
    | '<'
    | '<='
    | '<>'
    ;

arithmetic_expression
    : arithmetic_term
    | arithmetic_expression ('+' | '-') arithmetic_term
    ;

arithmetic_term
    : arithmetic_factor
    | arithmetic_term ('*' | '/') arithmetic_factor
    ;

arithmetic_factor
    : ('+' | '-')? arithmetic_primary
    ;

arithmetic_primary
    : atomic_valued_path_expression
    | numeric_literal
    | '(' arithmetic_expression ')'
    | single_valued_input_parameter
    | functions_returning_numerics
    | aggregate_expression
    | case_expression
    | function_invocation
    | subquery_expression
    ;

string_expression
    : atomic_valued_path_expression
    | string_literal
    | '(' string_expression ')'
    | single_valued_input_parameter
    | functions_returning_strings
    | aggregate_expression
    | case_expression
    | function_invocation
    | string_expression '||' string_expression
    | subquery_expression
    ;

datetime_expression
    : atomic_valued_path_expression
    | single_valued_input_parameter
    | functions_returning_datetime
    | special_datetime_expression
    | aggregate_expression
    | case_expression
    | function_invocation
    | subquery_expression
    ;

boolean_expression
    : atomic_valued_path_expression
    | special_boolean_expression
    | single_valued_input_parameter
    | case_expression
    | function_invocation
    | subquery_expression
    ;

enum_expression
    : atomic_valued_path_expression
    | enum_literal
    | single_valued_input_parameter
    | case_expression
    | subquery_expression
    ;

entity_expression
    : entity_valued_path_expression
    | single_valued_input_parameter
    ;

entity_type_expression
    : type_discriminator
    | entity_type_literal
    | single_valued_input_parameter
    ;

type_discriminator
    : 'TYPE' '(' entity_valued_path_expression ')'
    ;

functions_returning_numerics
    : 'LENGTH' '(' string_expression ')'
    | 'LOCATE' '(' string_expression ',' string_expression (',' arithmetic_expression)? ')'
    | 'ABS' '(' arithmetic_expression ')'
    | 'CEILING' '(' arithmetic_expression ')'
    | 'EXP' '(' arithmetic_expression ')'
    | 'FLOOR' '(' arithmetic_expression ')'
    | 'LN' '(' arithmetic_expression ')'
    | 'SIGN' '(' arithmetic_expression ')'
    | 'SQRT' '(' arithmetic_expression ')'
    | 'MOD' '(' arithmetic_expression',' arithmetic_expression ')'
    | 'POWER' '(' arithmetic_expression',' arithmetic_expression ')'
    | 'ROUND' '(' arithmetic_expression',' arithmetic_expression ')'
    | 'SIZE' '(' collection_valued_path_expression ')'
    | 'INDEX' '(' identification_variable ')'
    | 'EXTRACT' '(' datetime_field 'FROM' datetime_expression ')'
    | 'CAST' '(' string_expression 'AS' ('INTEGER' | 'LONG' | 'FLOAT' | 'DOUBLE') ')'
    ;

functions_returning_datetime
    : 'EXTRACT' '(' datetime_part 'FROM' datetime_expression ')'
    ;

functions_returning_strings
    : 'CONCAT' '(' string_expression ',' string_expression (',' string_expression)* ')'
    | 'SUBSTRING' '(' string_expression ',' arithmetic_expression (',' arithmetic_expression)? ')'
    | 'TRIM' '(' (trim_specification? trim_character? 'FROM')? string_expression ')'
    | 'LOWER' '(' string_expression ')'
    | 'UPPER' '(' string_expression ')'
    | 'CAST' '(' scalar_expression 'AS' 'STRING' ')'
    ;

trim_specification
    : 'LEADING'
    | 'TRAILING'
    | 'BOTH'
    ;

function_invocation
    : 'FUNCTION' '(' function_name (',' scalar_expression)* ')'
    ;

special_boolean_expression
    : 'TRUE'
    | 'FALSE'
    ;

special_datetime_expression
    : 'LOCAL' 'DATE'
    | 'LOCAL' 'TIME'
    | 'LOCAL' 'DATETIME'
    ;

entity_id_or_version_function
    : 'ID' '(' entity_valued_path_expression ')'
    | 'VERSION' '(' entity_valued_path_expression ')'
    ;

case_expression
    : general_case_expression
    | simple_case_expression
    | coalesce_expression
    | nullif_expression
    ;

general_case_expression
    : 'CASE' when_clause+
      ('ELSE' scalar_expression)?
      'END'
    ;

when_clause
    : 'WHEN' conditional_expression
      'THEN' scalar_expression
    ;

simple_case_expression
    : 'CASE' case_operand simple_when_clause+
      ('ELSE' scalar_expression)?
      'END'
    ;

case_operand
    : atomic_valued_path_expression
    | type_discriminator
    ;

simple_when_clause
    : 'WHEN' scalar_expression
      'THEN' scalar_expression
      ;

coalesce_expression
    : 'COALESCE' '(' scalar_expression (',' scalar_expression)+ ')'
    ;
```

```
nullif_expression
    : 'NULLIF' '(' scalar_expression ',' scalar_expression ')'
    ;

identification_variable : IDENTIFIER;

result_variable : IDENTIFIER;


entity_field
    : IDENTIFIER
    ;

embedded_field
    : IDENTIFIER
    ;

atomic_field
    : IDENTIFIER
    ;

collection_field
    : IDENTIFIER
    ;

structure_field
    : embedded_field
    | entity_field
    ;


datetime_field
    : IDENTIFIER
    ;

datetime_part
    : IDENTIFIER
    ;


entity_name : IDENTIFIER;

subtype : entity_name;

entity_type_literal : entity_name;


constructor_name : IDENTIFIER;


function_name : IDENTIFIER;


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

collection_valued_input_parameter : input_parameter;

single_valued_input_parameter : input_parameter;


literal : string_literal | numeric_literal;

numeric_literal : INTEGER | DOUBLE;

string_literal : STRING;

enum_literal : IDENTIFIER ('.' IDENTIFIER)*;
```

```
trim_character : CHARACTER;

escape_character : CHARACTER;

literal_pattern : STRING;
```