Encodo Systems AG – Garnmarkt 1 – 8400 Winterthur
Telephone +41 52 511 80 80 – www.encodo.com

Encodo QQL Handbook

Quino Query Language Specification

Abstract

The Quino Query Language (QQL) defines a syntax and semantics for formulating data requests against hierarchical data structures. It is easy to read and learn both for those familiar with SQL and non-programmers with a certain capacity for abstract thinking (i.e. power users). Learning only a few basic rules is enough to allow a user to quickly determine which data will be returned by all but the more complex queries. As with any other language, more complex concepts result in more complex texts, but the syntax of QQL limits these cases.

Authors

Marco von Ballmoos
Stephan Hauser

Table of Contents

Version History

Version	Date	Author	Comments
1.0	15.10.2011	mvb	Initial Version
1.1	16.12.2011	mvb	Integrated updates by sh
1.2	02.05.2012	mvb	Minor formatting and pagination fixes
1.3	26.09.2012	mvb	Integrated updates by rvb
1.4	01.11.2012	mvb	Integrated updates by rvb/sh/mvb
1.5	05.11.2012	Mvb	Finalized semantics for related lists; finalized syntax for grouping, sub-queries and sub-objects

Referenced Documents

Nr./Ref.	Document	Version	Date
[1]	MSDN Documentation for `String.Format()`: <http://msdn.microsoft.com/en-us/library/fht0f5be.aspx >		
[2]	Documentation for the ISO 8601 date/time format: <http://en.wikipedia.org/wiki/ISO_8601>		
[3]	MSDN Documentation for regular expressions <http://msdn.microsoft.com/en-us/library/hs600312(v=vs.71).aspx>		

Open Issues

Nr./Ref.	Document	Version	Date

Terms and Abbreviations

Term / Abbreviation	Definition / Explanation
Execution engine	Software that interprets a *query* and returns matching results from a *hierarchical data structure*
Hierarchical data structure	Data that is structured into a fixed hierarchy that conforms to a given *model*. Also called *data*.
Query	A text that conforms to QQL syntax and a given *model*
Model	Describes the structure of an application domain. Models are composed of *entities* and *relations* between them. Also called *metadata*.
Entity	Describes the shape of data in the structure using *properties* and *relations* to other entities. Also called a *metaclass* or *class*.
Relation	Describes a relationship between two entities, specifying the source *entity*, the target *entity* and the *cardinality* of each. A `Person` may have a relation to `TimeEntry` called `TimeEntries`.

Term / Abbreviation	Definition / Explanation
Cardinality	Describes the number of entities that can be added to one side of a *relation*. In the `TimeEntries` relation given above, the `Person` side has a cardinality of 1, which means that each `TimeEntry` *must* be associated with a person. The `TimeEntry` side has a cardinality of *n*, which means that a `Person` may have zero or more `TimeEntry` entities associated with it.
Property	Describes a single piece of data in an entity. For example a `Person` may have the properties `FirstName` and `BirthDate`. A property has a fixed *type*.
Loadgroup	Describes a list of *properties* and/or *relations* for a *metaclass*. Each metaclass has a default loadgroup comprising all persistent properties.
Type	Defines the valid data that can be assigned to a *property*. For example, the two properties named in the example above, `FirstName` and `BirthDate` would have the types `Text` and `Date`, respectively.
Scalar type	A type that is a single value, not a list of values or objects (e.g. a number, a string, etc.)
Infix operator	An operator that accepts two arguments, referred to as the left and right arguments.
Root expression	An expression that appears directly within a section.

1 Introduction

QQL defines a syntax and semantics for writing queries against *hierarchical data structures*. A query describes a set of data by choosing an initial *context* (see "3 – Context & Scopes") in the data and specifying which data are to be returned (see "4.3 – Selection" and "4.5 – Filtering") and how the results are to be organized (see "4.6 – Ordering" and "5 – Grouping Queries"). An *execution engine* generates this result by applying the query to the data (see "6 – Evaluation").

1.1 Goals

- **Expressive:** QQL must be expressive enough to unambiguously capture the intent of requests that a typical application is likely to make.
- **Reproducible:** The result of executing a query against a set of data must be consistent, reproducible and testable.
- **Readable:** Though clearly a secondary goal, readability is very important. The intent of a query should be "obvious"—or as clear as possible for a given level of complexity.
- **Programmable:** And finally, it should be easy to create, combine and format query texts programmatically.

1.1.1 Man vs. Machine

Given these goals, QQL has certain constructs that are more appropriate for human authors and others that are more appropriate for machines. This document distinguishes between that which is *valid* and that which is *recommended*. See "10 – Best Practices" for examples.

1.1.2 Expressiveness and Performance

A query precisely describes the results that an application would like to receive. Therefore, it is possible for an application to make requests for which it is logically difficult—or impossible—to generate results efficiently. Examples and further discussion can be found in "11.1 – General Execution".

1.2 Structure

This handbook takes a top-down approach to learning QQL. It starts with examples to familiarize the reader with syntax and the types of requests QQL can make and continues with a discussion of the high-level structures, evaluation and resolution algorithms and finally, syntax, basic types, operators and library functions.

1.3 Target Audience

With clarity high on the list of priorities when designing the syntax, the target audience for QQL is both those already familiar with other query languages (e.g. **SQL**) *and* non-programmers with a certain capacity for abstract thinking (i.e. power users).

2 Examples

The model used in the examples in this document describes a time-tracking application called *Punchclock*[1]. It contains entities like companies, people, time-entries, projects and customers. This context is used to aid understanding by providing an at least somewhat familiar real-world model.

This section includes several examples intended to show what QQL looks like for different types of queries. As mentioned in the introduction, the syntax focuses on clarity and ease of construction, so the examples should be relatively intuitive. See "7 – Syntax" for more information on language constructs.

2.1 Simple Standard Query

The following query returns the first and last name of all active people as well as their 10 most recent time entries, reverse-sorted first by last name, then by first name.

```
Person
{
  select
  {
    FirstName; LastName;
    Sample:= TimeEntries { orderby Date desc; limit 10 }
  }
  where Active
  orderby
  {
    LastName desc;
    FirstName desc;
  }
}
```

2.2 Intermediate Standard Query

This query builds on the simple example to include more information per person and filter time entries and people. The example also shows a standard function, `Year`, as well as an aggregated function, `Count`. See "9.2 – Date" and "9.4 – Aggregation" for more information.

```
Person
{
  select
  {
    FirstName; LastName;
    ShortContact:= ContactInfo { ZipCode }
    Contracts;
    Sample:= TimeEntries
    {
      Date; Amount;
      where Date.Year = 2009;
      orderby Date desc;
      limit 10
    }
  }
  where
  {
    Active;
    TimeEntries.Count > 10000;
  }
  orderby
  {
    LastName desc;
    FirstName desc;
  }
}
```

[1] This is, incidentally, a real application that Encodo built using Quino for time-tracking, reporting and invoicing.

2.3 Complex Standard Query

This more complex query builds on the intermediate example to introduce variables. Those that are used in multiple sections are declared in the `var` section.

```
Person
{
  var
  {
    yearsEmployedCount:= (Now - EmploymentDate).Years;
    bigContracts:= Contracts { where Amount > 500000 }
  }
  select
  {
    FirstName; LastName;
    ShortContact:= ContactInfo { ZipCode }
    Contracts;
    TotalContractAmount:= Contracts.Amount.Sum;
    Sample:= TimeEntries
    {
      Date; Amount;
      where Date.Year = 2009;
      orderby Date desc;
      limit 10
    }
  }
  where
  {
    Active;
    TimeEntries.Count > 10000;
    yearsEmployedCount > 10 or IsManager;
    bigContracts.Count > 2;
  }
  orderby
  {
    bigContracts.Count;
    yearsEmployedCount;
    LastName desc;
    FirstName desc;
  }
}
```

2.4 Simple Grouping Query

The following query groups active people by last name and returns the age of the youngest person and the maximum contracts for each last name. Results are ordered by the maximum contracts for each group and then by last name.

```
group Person
{
  groupby LastName;
  select
  {
    default;
    Age:= (Now - BirthDate.Min).Year;
    MaxContracts:= Contracts.Count.Max
  }
  where Active;
  orderby
  {
    MaxContracts desc;
    LastName desc;
  }
}
```

For more information about valid expressions for the various sections, see "5 – Grouping Queries".

2.5 Complex Grouping Query

This query builds on the intermediate example by calculating maximum employment time returning only groups for people employed 5 years or more. The query also returns the grouped items using a sub-query (the sub-query was copied from "2.1 – Simple Standard Query").

```
group Person
{
  groupby
  {
    LastName;
    var minYearsEmployed:= (Now - EmploymentDate.Min).Years;
  }
  select
  {
    default;
    Age:= (Now - BirthDate.Min).Year;
    MaxContracts:= Contracts.Count.Max
  }
  where Active;
  having
  {
    LastName beginswith "C";
    minYearsEmployed >= 5;
  }
  orderby
  {
    minYearsEmployed desc;
  }
  selectObjects
  {
    select
    {
      FirstName; LastName;
      Sample:= TimeEntries { orderby Date desc; limit 10 }
    }
    where
    {
      IsManager;
    }
    orderby
    {
      LastName desc;
      FirstName desc;
    }
  }
}
```

2.6 Standard Query with Grouping Query

The following is a simple query that returns all people with default properties as well as time-entries from the last month grouped by day.

```
Person
{
  where Active
  orderby
  {
    LastName desc;
    FirstName desc;
  }
  group TimeEntries
  {
    groupby Date;
    objects
  }
}
```

2.7 Nested Grouping Queries

The following query groups people first by `LastName`, then by `FirstName` and then returns the 10 most recent `TimeEntries` as well as the `FirstName` for each person in each second-level group.

```
group Person
{
  groupby LastName;
  group objects
  {
    groupby FirstName;
    objects
    {
      Sample:= TimeEntries { orderby Date desc; limit 10 }
    }
  }
}
```

3 Context & Scopes

A query text does not stand alone; it only makes sense within a certain *context*, defined by the *model*. A model consists of *entities*, which have *properties* and *relations* to other entities. The data against which the query is executed must conform to the same model.

There are two types of queries, *Standard* and *Grouping*. As shown in the examples, these queries can be nested; the level of nesting is only limited by the model underlying the query.[2] A query contains keywords that include new scopes in the context.

A scope contains a list of available identifiers. A context is the union of one or more scopes. The query-type, section-type and the context taken together determine which identifiers can be referenced and which identifiers can be introduced (i.e. by creating a variable). See "6.8 – Resolving Identifiers" for more information.

3.1 Global Scope

The global scope contains identifiers corresponding to global functions and namespaces that contain other functions. Identifiers in this scope are always available.

3.2 Model Scope

A scope based on a model includes identifiers for the metaclasses for that model. This scope is implicit, defaulting to the default model for the execution engine. A query may reference a model explicitly by prepending the name of the model to the initial metaclass scope.

The following example selects all people from the *Punchclock* model.

```
Punchclock.Person
```

As mentioned above, an explicit model scope is optional and will not be used in the remaining queries in this document.

3.3 Metaclass Scopes

A scope based on a metaclass includes identifiers for the properties and relations for that metaclass. However, only the identifiers of the *current* metaclass scope (i.e. the topmost metaclass scope on the stack) are directly available at a given position in the query.

Identifiers from outer metaclass scopes are available through relations on the current metaclass or through the *Predecessor* relation (see "6.3 – Predecessor" for more information).

3.3.1 Sections

A metaclass scope may contain zero or more sections. If the metaclass was preceded by the keyword `group`, the scope may contain the sections valid for a grouping query; otherwise, the scope may contain the sections for a standard query. See "4 – Standard Queries" and "5 – Grouping Queries" for details about the available sections.

[2] The execution engine may also limit the nesting level. See "11.1 – General Execution" for more information.

The following rules apply to sections:

- All sections are optional
- The order of the sections is not relevant
- A given type of section may appear more than once (e.g. a given block may have multiple "`where`" sections)
- Any combination of lines and blocks is allowed (see "7.1 – Lines" and "7.2 – Blocks" for more information).
- Regardless of how the sections are included in the query text, the query will be normalized before execution. See "6.5 – Normalizing Queries" for more information.

The validity constraints described above are deliberately weak in order to accommodate machine-generated queries or queries that have been combined from various sources.

See "10.7 – Consistent Declaration" to see more examples of the recommended declaration style when writing queries by hand.

3.4 Initial Metaclass

The first identifier of a query—or the last identifier in the dot-separated chain, if a model namespace is specified—must correspond to a metaclass from the model; this defines the initial metaclass scope in the context.

3.5 Relations

The other way to introduce a metaclass scope is by including a relation from the current metaclass. A metaclass scope for the target class of the relation will be added to the context.

When the *block* or *line* for a relation is terminated, the corresponding scope is popped from the context (see "7.1 – Lines" and "7.2 – Blocks").

3.6 Variables

Variables are added to the current scope and are available as long as that scope is on the stack. Unlike metaclass identifiers, which are only directly available in the same scope, variables are available to all nested scopes, as well.

See "6.1 – Variables and Scopes" for more information and examples.

4 Standard Queries

A query is broken into sections that provide information about projection (data-selection), ordering, filtering and so on. This chapter describes the sections for standard queries. See "5 – Grouping Queries" to learn about grouping queries.

4.1 Special Keywords

The following keywords correspond to macros that are valid in the `select`, `distinct` and `orderby` sections.

Symbol	Usage / Meaning
`primary`	A keyword that refers to the set of properties that comprise the primary key of the metaclass for the current scope
`default`	A keyword that refers to the elements of the default loadgroup
`properties`	A keyword that refers to all properties of the metaclass for the current scope
`relatedobjects`	A keyword that refers to all relations of the metaclass for the current scope having a target cardinality of one
`relatedlists`	A keyword that refers to all relations of the metaclass for the current scope having a target cardinality of greater than one

In addition, the `property` and `loadgroup` operators can also be used to disambiguate references. See "6.8 – Resolving Identifiers" for more information. Keyword/identifier collisions can be resolved using the @-syntax; see "7.4 – Identifiers" for more information.

4.2 Variables

A variable section is introduced by the `var` keyword and can contain only variable assignments (described in more detail in "6.1 – Variables and Scopes"). Though variables can be declared in any section, those declared in this section have no other effect on the query (i.e. selection, ordering, etc.). This section acts as a scratchpad for the scope in which it is declared.

The variables in this section can be used anywhere in any other query section in the same scope (e.g. a *select* block or line) as well as anywhere in a nested scope.

4.3 Selection

A selection section is introduced by the `select` keyword. Expressions in this section indicate which data to include in the result for a metaclass scope.

Identifiers in this section are resolved according to the rules outlined in"6.8 – Resolving Identifiers".

4.3.1 Default Selection

If a metaclass scope does not include any *select* sections, the elements of the default loadgroup are selected by default. If, however, any properties or relations are explicitly selected, the query returns only that data. To explicitly include the default properties, use the `default` keyword.

The following query returns the default properties for people as well as contracts with default properties.

```
Person
{
  select { default; Contracts }
}
```

4.3.2 Select All

By default, a query returns the properties from the default loadgroup of a metaclass, which may exclude some properties. In order to include all scalar (non-relational) properties of a metaclass, use the `properties` keyword.

For example, a person has a property called `Picture` that is not in the default loadgroup. The following query will include that property, as well as any others that are not in the default loadgroup.

```
Person
{
  select { properties }
}
```

4.3.3 Ordering of expressions

Each row in the result set includes values for the selected expressions in the order that they were declared in the query. If `properties` or `default` is used, the expressions are included in the order in which the properties and relations are declared in the metaclass that defines the scope.

The following query returns the birth date, last name and then first name.

```
Person
{
  FirstName;
  orderby { BirthDate }
  LastName;
  orderby { FirstName }
  select { BirthDate }
}
```

4.3.4 Omitting the 'select' Keyword

The select section is the default section when a metaclass scope is created. Any expressions that appear outside of a specific section (e.g. `where`, `orderby`, etc.) are added to the selection. The following query is equivalent to the query declared in "4.3.1 – Default Selection" above:

```
Person
{
  default; Contracts
}
```

4.4 Distinct

A distinct section is introduced by the `distinct` keyword. Expressions in this section determine the data which must be unique or "distinct" from all other rows in order to be included in the result. The values for non-distinct expressions in the selection are taken from the first row in each group, so ordering is important when working with distinct queries.

The following query orders people by last name and then by first name and returns the first and last name for each person for each unique birth date (i.e. if John Adams and Bob Jenkins both have the same birthday, then only John Adams is returned).

```
Person
{
```

```
  distinct { BirthDate }
  LastName; FirstName;
  orderby { LastName; FirstName }
}
```

4.4.1 Default/empty distinct

If the distinct section is included but is empty, all expressions included in the query selection are treated as if they were declared in the distinct section. This behavior not only matches the common behavior for most SQL databases, it makes it easier to make a query distinct.

The following query returns the first name of the oldest person with that first name:

```
Person
{
  distinct;
  FirstName;
  orderby { BirthDate; }
}
```

This style can be most useful for filtering unexpected (and unwanted) duplicates from a result.

4.4.2 Distinct with default selection

If both the distinct section is included but is empty *and* the select section is not explicit (i.e. is included but is empty or is explicitly declared as including the `default`), the query returns all results that are distinct in all properties in the default loadgroup except for those properties that comprise the primary key (because a query that includes the primary key in the `distinct` section is equivalent to a non-distinct query).

The following query returns all people who are distinct from one another in the default properties for a person.

```
Person
{
  distinct
}
```

4.4.3 Custom Restrictions

The following query returns the properties from the default loadgroup for all people who are distinct from one another in last name and first name.

```
Person
{
  distinct { LastName; FirstName }
}
```

This section may also refer to or define variables. The query below returns the properties from the default loadgroup for each person who is distinct from all other people in last name and number of contracts.

```
Person
{
  distinct { LastName; Contracts.Count }
}
```

A distinct section may also use the `default` and `primary` keywords, though including the primary key in a distinct section is equivalent to using a non-distinct query. The following query returns the first and last names of all people who are distinct from one another in all the properties from the default loadgroup except for `MiddleName` and `BirthDate`.

```
Person
{
  distinct { default - [MiddleName, BirthDate] }
}
```

Variables can be declared in this section, but doing so is not recommended. See "10.1 – Defining Variables" for more information.

4.4.4 Compared to a Grouping Query

Any query that includes a `distinct` section can be rewritten as a grouping query.

Consider the following simple example that returns the default properties of all people for each unique combination of `FirstName` and `LastName`.

```
Person
{
  distinct { LastName; FirstName }
}
```

The query above implicitly selects all the properties in the default loadgroup. A grouping query does not automatically select these properties, so the equivalent grouping will have to do so explicitly. The following query shows the equivalent grouping query for the query above.

```
group Person
{
  groupby { LastName; FirstName }
  select
  {
    LastName; FirstName;
    BirthDate.First; EmploymentDate.First;
    primary.First;
    // other properties in the default loadgroup
  }
}
```

Using a distinct query is much more concise and intuitive than the equivalent grouping query.

4.4.5 Implications for Performance

Whereas almost all SQL databases support the default distinct behavior, some do not support restricting to a distinct set of expressions. See "11.1 – General Execution" for performance implications.

4.5 Filtering

A filtering section is introduced by the `where` keyword. The expressions in this section comprise a filter for data in the current scope. Expressions are evaluated as Booleans in declaration order; any data for which *all* expressions yield true are included in the result. Boolean short-circuiting is in effect, so once an expression returns false for a row, that row is excluded from the result without evaluating any further filters.

The following additional keywords are supported in these sections.

Symbol	Usage / Meaning
`not empty`	A keyword that can be included only within a metaclass scope corresponding to a relation. If present, data is returned only if the relation contains at least one element.
`empty`	A keyword that can be included only within a metaclass scope corresponding to a relation. If present, data is returned only if the relation contains no elements.

Unlike the `orderby` section, the default restrictions and filters in the metaclass are *always* included and cannot be overridden by the query. The only way to avoid including the default filters for a metaclass is to use an ancestor that does not have the filter.

The following query returns all people that have at least one time-entry but no contracts.

```
Person
{
  where TimeEntries not empty;
  where Contracts empty;
}
```

4.6 Ordering

An ordering section is introduced by the `orderby` keyword. The expressions in this section determine the order of the data in the current scope. See "11.3 – Ordering Data" for more information on how ordering is implemented by an execution engine. The expressions are applied in the order that they are declared in the query, unless the `pos`-keyword is used (as explained in "4.6.2 – Ordering").

The following additional keywords are supported in these sections.

Symbol	Usage / Meaning
`asc`	A suffix operator that indicates that the expression sorts data from lowest to highest
`desc`	A suffix operator that indicates that the expression sorts data from highest to lowest
`pos`	An infix operator that sets the position in the ordering of the expression on the left-hand side to the value in the right-hand side. The right-hand side can only be a constant integer value.
`default`	A keyword that represents the set of zero or more expressions that are the default ordering in the metaclass for the current scope

4.6.1 Default Ordering

The default order for elements is determined by the metadata for the current scope. If that metadata does not provide an explicit ordering, elements are ordered ascending by primary key.

Assume that people are sorted by `LastName` and then by `FirstName` by default. The following query with no explicit ordering returns people in that default order.

```
Person
```

If a query includes an explicit ordering, the default ordering is no longer included. The following query returns all people sorted by `BirthDate`.

```
Person
{
  orderby { BirthDate }
}
```

To include the default sorting in a query, use the `default` keyword. The following example returns all people sorted by `BirthDate`, then by `LastName` and `FirstName`.

```
Person
{
  orderby { BirthDate; default }
}
```

The default ordering for elements can be explicitly ignored by including an empty `orderby` section. The following query returns people ordered by primary key ascending rather than by the default ordering by `LastName` then `FirstName`.

```
Person
```

```
{
  orderby { }
}
```

4.6.2 Ordering Priority

As in the `select` section, declaration order matters in the `orderby` section. Whereas it's relatively easy to control the declaration order in queries written by hand, machine-generated queries or queries combined from multiple sources may end up with the wrong logical ordering.

For example, assume the following two statements must be combined into a query for people.

The first query selects the first and last name and sorts by first name:

```
Person
{
  FirstName;
  LastName;
  orderby { FirstName }
}
```

The second query selects the birthdate and sorts by it:

```
Person
{
  BirthDate;
  orderby { BirthDate }
}
```

If these two queries are combined, we get the following query, which selects `FirstName`, `LastName` and `BirthDate` and sorts first by `FirstName`, then by `BirthDate`.

```
Person
{
  FirstName;
  LastName;
  orderby { FirstName }
  BirthDate;
  orderby { BirthDate }
}
```

However, what if the actual intent of adding the second query is to not only add the field, but also to sort *primarily* by the `BirthDate`? The second query can signal this intent by specifying a priority for the ordering. Priorities are integer values and follow these rules:

- The priority value can be any 32-bit integer value
- The default priority is `0`
- Orderings are applied in descending priority order and then by declaration order
- Use a higher priority value to force an ordering to be applied first

In order to sort primarily by `BirthDate`, use a priority as shown below.

```
Person
{
  BirthDate;
  orderby { BirthDate priority 1 }
}
```

When combined with the first query, this yields the following text:

```
Person
{
  FirstName;
  LastName;
  orderby { FirstName }
  BirthDate;
  orderby { BirthDate priority 1 }
}
```

Since the ordering by `FirstName` has an implicit priority of `0`, the ordering by `BirthDate` will be applied first, as shown below.

```
Person
{
  FirstName;
  LastName;
  BirthDate;
  orderby { BirthDate }
  orderby { FirstName }
}
```

4.6.3 Ordering of Nulls

Null values are considered to be larger than non-null values. That is, null values are sorted *last* when `asc` is specified or implied and sorted first when `desc` is specified.

4.7 Paginating and Limiting Results

A metaclass scope can control which of all possible matching objects should be returned with two keywords, `offset` and `limit`. This is especially useful for paginating or limiting result sets with many objects.

Symbol	Usage / Meaning
`limit`	A prefix operator that indicates the maximum number of results to return for the current scope. The expression can only be a constant integer value.
`offset`	A prefix operator that indicates the 0-based index of the first result to return from the list of objects that match the query. The expression can only be a constant integer value.

The following rules apply for these keywords:

- They can only appear immediately within the block that defines the metaclass scope
- They may appear more than once, but only the last instance of each is used
- They can only be written as lines and the arguments can only be constant integers

The following recommendations apply for these keywords in queries written by hand:

- They should be included at most once
- They should appear in the following order: `limit`, `offset`
- They should appear either at the very beginning or the very end of a query

The following example returns the default properties for the first 10 people in a list sorted by `BirthDate`.

```
Person
{
  orderby BirthDate;
  limit 10
}
```

The following example returns the default properties for 10 people starting with the 50th person in a list sorted by `BirthDate`.

```
Person
{
  orderby BirthDate;
  offset 50; limit 10
}
```

And finally, the following example returns the first 10 people in a list sorted by `BirthDate` as well as the first 20 time entries for each person.

```
Person
{
  default;
  Sample:= TimeEntries { limit 20 }
  orderby BirthDate;
  limit 10
}
```

Limiting or offsetting output for a sub-query has potential performance implications; see "11.1 – General Execution" for more information.

5 Grouping Queries

A grouped query returns multiple rows, each of which is a "group" that represents the aggregated data for all objects that share the same values for certain expressions (called "grouping expressions").

Aggregation functions apply to all objects within each group, producing a separate value for each group rather than a single value for the entire result.

The grouping query returns information about the group but may also include a sub-query that describes the content of each group to return as well. See "5.9 – Selecting Objects for each Group".

5.1 The 'group' Keyword

A grouped query is introduced by the `group` keyword. It applies either to the initial metaclass or to any relation (though it only makes sense for relations that have target cardinality greater than one). It can also appear immediately after the `objects` keyword to indicate that the sub-objects of a group are also grouped; see "5.9 – Selecting Objects for each Group" for more information.

5.2 Variables

A variable section is introduced by the `var` keyword. This section has the same semantics as in a standard query (e.g. variables are not included in the selection); see "4.2 – Variables" for more information.

The sections that restrict which expressions may be used (`select`, `orderby` and `having`) may only use variables that conform to those restrictions.

The following query returns people grouped by the number of contracts they have, returning that number as well as the number of people in each group, but only for people with 3 or more contracts.

```
group Person
{
  var ContractCount:= Contracts.Count;
  groupby ContractCount;
  select { ContractCount; Count };
  having ContractCount >= 3;
  orderby ContractCount
}
```

Note that the untargeted call to `Count` above applies to the group in which it appears (see "9.4.1 – Untargeted Aggregations" for more information).

5.3 Grouping Expressions

A grouping section is introduced by the `groupby` keyword. Expressions in this section define the values that must match in order for objects to be grouped together. A grouping expression must be a scalar value or a relation with target cardinality of one or an aggregation function over a relation with multiple target cardinality.

A grouping query must include at least one grouping expression.

5.3.1 Grouping by Scalar Value

The following query returns all people with their 100 most recent time-entries grouped by date.

```
Person
{
  default;
  GroupedTimeEntries:= group TimeEntries
  {
    groupby Date;
    orderby Date desc;
    limit 100;
    objects;
  }
}
```

5.3.2 Grouping by Object

The following query returns all people with their 100 most recent time-entries grouped by project.

```
Person
{
  default;
  GroupedTimeEntries:= group TimeEntries
  {
    groupby Project;
    limit 100;
    objects;
  }
}
```

5.3.3 Grouping by Multiple Values or Objects

The following query returns all people with their time-entries grouped by project and date and sorted by most recently used project.

```
Person
{
  default;
  GroupedTimeEntries:= group TimeEntries
  {
    groupby { Project; Date }
    orderby Date desc;
    objects;
  }
}
```

Since the `orderby` appears within the group, it is applied to the data *before* it is grouped.

5.4 Selection

A selection section is introduced by the `select` keyword. Expressions in this section indicate which data to include in the result for a metaclass scope.

A grouping query may only return grouping expressions or expressions that aggregate data from the group. The following example returns the `LastName` and total amount of time for each group.

```
group Person
{
  groupby LastName;
  select { LastName; TimeEntries.Time.Sum.Sum }
}
```

The expression `TimeEntries.Time.Sum` returns the total amount of time per person in the group; the second call to `Sum` returns the total amount of time for the group.

The following example gets the average amount of time for each group as well as the sum.

```
group Person
{
  groupby LastName;
  select
  {
    LastName;
    Total:= TimeEntries.Time.Sum.Sum;
    Average:= TimeEntries.Time.Sum.Average
  }
}
```

The query above seems quite straightforward, but will in all likelihood not scale well. See "11.1 – General Execution" for performance implications and "9.4 – Aggregation" for more information about aggregation functions.

5.4.1 Default Expressions

If there are no `select` sections, the grouping expressions are selected by default. The following query groups people by `LastName` and returns only that expression.

```
group Person
{
  groupby LastName;
}
```

A grouping query may also use the `default` keyword to select the grouping expressions explicitly. The following query selects the grouping expressions as well as total amount of time for all people in that group.

```
group Person
{
  groupby { LastName; FirstName }
  select { default; TimeEntries.Sum.Sum }
}
```

5.4.2 Returning Other Data

Grouping queries can include other data in the group, but it must be aggregated. An easy way to aggregate data is to use the `First` function and set the ordering to determine which object appears first in each group.[3]

The following example returns the `LastName` of the people in each group as well as the `BirthDate` and the total amount of time for the youngest member.

```
group Person
{
  groupby LastName;
  select
  {
    LastName;
    BirthDate.First;
    TimeEntries.Amount.Sum.First
  }
  orderby BirthDate.First desc;
}
```

Alternatively, a grouped query may also include the grouped items in the result; see "5.9 – Selecting Objects for each Group" for more information.

[3] Using the `distinct` section in a standard query does this automatically; see "4.4.4 – Compared to a Grouping Query" for more information.

5.5 Filtering data before grouping

A filtering section is introduced by the `where` keyword. This section has the same semantics as in a standard query; see "4.5 – Filtering" for more information.

This section restricts the data that is considered for grouping and is applied *before* grouping. The following example groups people by last name, but only for managers.

```
group Person
{
  groupby LastName;
  where IsManager;
}
```

5.6 Filtering grouped data

A group filtering section is introduced by the `having` keyword. This section may only include grouping expressions or aggregation expressions that reference data from the group.

The following query groups people by `LastName` and returns only the groups where the birthdate of the youngest person with that last name is after 1990.

```
group Person
{
  groupby LastName;
  having BirthDate.Max.Year > 1990;
}
```

5.7 Ordering

An ordering section is introduced by the `orderby` keyword. This section may only include grouping expressions or aggregation expressions that reference data from the group.

The following query groups people by `LastName` sorted by the birthdate of the youngest person with that last name.

```
group Person
{
  groupby LastName;
  orderby BirthDate.Max;
}
```

An ordering that includes expressions

TODO: Figure out how to apply before and after

Ordering does not need to be applied before because the grouping affects the ordering anyway. To "fake" an ordering, use Min and Max.

The following query selects grouped by `LastName`

5.8 Pagination and Limiting Results

Pagination and limiting results has the same semantics as in a standard query.

The following query returns all people with the project, date and time-entries for the most recently-used project.

```
Person
{
  default;
  GroupedTimeEntries:= group TimeEntries
  {
    groupby { Project; Date }
    orderby Date desc;
    limit 1;
    objects;
  }
```

```
}
```

The limit above applies to the number of groups returned. Use a second limit to restrict the number of time-entries returned in the selected objects to 10, as shown below.

```
Person
{
  default;
  GroupedTimeEntries:= group TimeEntries
  {
    groupby { Project; Date }
    orderby Date desc;
    limit 1;
    objects { limit 10; }
  }
}
```

See "5.8 – Pagination and Limiting Results" for more information.

5.9 Selecting Objects for each Group

The keyword `selectObjects` introduces a sub-query for the grouped items. The metaclass scope for the sub-query is the same as that for the enclosing grouping query.

The example below shows a query that groups people by `LastName` and also returns the `FirstName`, `LastName` and 10 most recent `TimeEntries` for each person in each group.

```
group Person
{
  groupby LastName;
  objects
  {
    FirstName, LastName, TimeEntries { orderby Date desc; limit 10 }
  }
}
```

Since there is no explicit `select` section, the grouping query returns the `LastName` for each group.

5.9.1 Name of the "objects" Relation

The following rules determine the default name of the relation:

- Use the variable name if possible
- Use the name of the relation if the query addresses a relation
- Use "Objects"

In the result set for the query below, the sub-objects for each group are accessible through the identifier "Employees" instead of the default "Objects".

```
group Person
{
  groupby LastName;
  Employees:= objects
  {
    FirstName, LastName, TimeEntries { orderby Date desc; limit 10 }
  }
}
```

5.9.2 Default Sub-queries

The sub-query section may also be written as a line instead of a block if only defaults are used. The query below returns people grouped by `LastName`, selecting the default properties for all of the people in each group.

```
group Person
{
  groupby LastName;
  objects;
}
```

5.9.3 Groups within Groups

The sub-query may also be a grouped query. Even when the sub-query is a grouping query, the identifier is optional, as shown in the example below, which groups people first by `LastName`, then by `FirstName` and then returns the 10 most recent `TimeEntries` for each person in each second-level group.

```
group Person
{
  by LastName;
  objects group
  {
    by FirstName;
    objects
    {
      TimeEntries { orderby Date desc; limit 10 }
    }
  }
}
```

In this case, the both the relations defined by `selectObjects` are called "People", which is valid because they are nested.

Even such a simple-looking query has quite a complex structure. In pseudo-code, an application would reference the first time-entry in the result set using something like the following:

```
Groups[0]['People']['People'][0].
```

5.9.4 Multiple `selectObjects` Relations

A query may include multiple `selectObjects` sections, but only one may use the default name; others must be assigned to variables (see "6.1 – Variables and Scopes").

Variables provide a simple way to return the contents of a relation multiple times with different sub-queries. The following example returns a person's time-entries as three different sub-queries.

```
Person
{
  TimeByGroup:= group TimeEntries { groupby Project; objects }
  TimeToday:= TimeEntries { where Date = Now }
  TimeThisMonth:= TimeEntries
  {
    where Date.Year = Now.Year and Date.Month = Now.Month
  }
}
```

The following example returns two views of a customer's projects in the same result set.

```
Customer
{
  BigProjects:= Projects { where TimeEntries.Count > 1000 }
  SmallProjects:= Projects { where TimeEntries.Count < 10 }
}
```

The following query returns people grouped by last name, including default details for people in each group under the default name "Objects" as well as only the first and last name for the people in each group under the name "NamesOnly".

```
group Person
{
  groupby LastName;
  objects;
  NamesOnly:= objects { FirstName; LastName }
}
```

6 Evaluation

The previous sections introduced the different types of queries and discussed higher-level rules for reading and writing them. This section includes details about variable declaration, identifier and function resolution as well as fallbacks and default in the query language.

For low-level details, see "7 – Syntax" and "8 – Data Types and Operators".

6.1 Variables and Scopes

Queries create variables by assigning an expression to an identifier (see "7.4 – Identifiers" and "7.8 – Assignment").

The following rules apply to variable declarations.

- **Types:** Any expression can be assigned to a variable, including scalar values, objects or queries/relations.
- **Scope:** A variable is accessible from anywhere in the metaclass scope in which it is declared and in any nested scopes.
- **Valid Names:** There are no uniqueness requirements for variable names; the name can match a metadata identifier or a variable name from an outer scope or a function name. Queries can use this feature to override identifiers from the default scope; see "6.8 – Resolving Identifiers" for examples.
- **Multiple Declarations:** If a variable is declared more than once, the last value assigned to it in the declaration order will be the value used when evaluating the query. In query written manually, it is not recommended to assign a variable more than once; see "10.1 – Defining Variables" for more information.

6.2 Referencing Variable in Outer Scopes

The following example shows a sub-query that references a variable declared in an outer scope. The query returns the default properties for a person, total number of time entries and all projects that contributed at least 10% of the time entries for that person.

```
Person
{
  default;
  TotalTimeEntryCount:= TimeEntries.Count;
  Projects
  {
    where TimeEntries.Count >= TotalTimeEntryCount / 10;
  }
}
```

6.3 Predecessor

A query can refer to metaclasses from outer scopes via variables defined in those scopes (as in the example above) or via named relations of the metaclass or the `predecessor` keyword.

The predecessor is a macro that refers to a relation with single cardinality that refers to the object that defined the metaclass scope immediately outside of this one. In the outermost scope, the predecessor returns null.

The following example includes the company's name as a property on each person returned in the sub-query.

```
Company
{
  People
  {
    default;
    CompanyName:= predecessor.Name;
  }
}
```

Referring to the predecessor is only *necessary* if the desired relation does not exist in the current metaclass scope. In the example, the `Person` already has a relation of single cardinality pointing to the `Company` to which it belongs, so it could have just used that instead.

```
Company
{
  People
  {
    default;
    CompanyName:= Company.Name;
  }
}
```

The predecessor can be assigned to a variable like the result of any other relation. This can be useful when a deeply nested sub-query needs to refer to fields in the outermost scope (for example).

The query can chain calls to predecessor until it reaches the desired scope, as in the example below.

```
Company
{
  People
  {
    TimeEntries
    {
      Project
      {
        matches:= Customer
        {
          where Company = predecessor.predecessor.predecessor.predecessor
        }
      }
    }
  }
}
```

Instead, the query could declare a variable at the outer scope and use that variable in the deeply nested scope, as shown below.

```
Company
{
  var mainCompany:= current;
  People
  {
    TimeEntries
    {
      Project
      {
        Customer
        {
          where Company = mainCompany
        }
      }
    }
  }
}
```

This example also makes use of the `current` keyword, which is discussed below.

6.4 Current

As already noted, a query can refer to identifiers in the current scope directly. A query may also refer to the current scope explicitly, using the `current` macro. Whereas it is recommended to omit the keyword where optional, it is useful for creating variables to be used in nested scopes, as shown in the final example in the previous section.

6.5 Normalizing Queries

Before a query is executed, it is normalized according to the following rules:

- For each metaclass scope, all sections of the same type are collected into a single section of that type.
- Expressions are kept in declaration order, which is important for the `select`, `where` and `orderby` sections.[4]
- All variable declarations are moved to the `var` section, even if used only once

[4] Whereas declaration order is obvious for the `orderby` section, the order in the `select` section determines the order of fields in the result set and can be used to optimize execution speed of a query by placing the most stringent expressions first in the `where` section.

The following example is from "10.7 – Consistent Declaration":

```
Person
{
  Contracts;
  limit 10;
  orderby [LastName, FirstName];
  where FirstName contains 'M';
  where LastName contains 'M';
  select FirstName;
  select Lastname;
  orderby BirthDate pos 0;
  where ContractCount < 10;
  offset 2;
  var ContractCount:= Contracts.Count;
  select [ContractCount, MiddleName, BirthDate];
}
```

The normalized form of this query is:

```
Person
{
  var
  {
    ContractCount:= Contracts.Count;
  }
  select
  {
    Contracts; FirstName; LastName; ContractCount; MiddleName; BirthDate;
  }
  where
  {
    FirstName contains 'M';
    LastName contains 'M';
    ContractCount < 10;
  }
  orderby
  {
    BirthDate; LastName; FirstName;
  }
  offset 2; limit 10;
}
```

6.6 Root Expressions & Identifiers

A root expression is an expression that appears directly within a block; in the example below, `FirstName`, `LastName`, `FullName` and `Total` are root expressions.

```
Person
{
  FirstName;
  LastName;
  FullName:= "{LastName}: {Company.Name}";
  Total:= 45 + 23 * 5;
}
```

Some sections require that the identifiers for root expressions within those sections be unique. Other sections do not require uniqueness but still use the identifier to uniquely identify root expressions in that section.

The following sections require that all root expression identifiers are unique:

- `select`
- `distinct`
- `groupby`

To prevent queries from having to explicitly assign names to expressions, there is an algorithm by which an *implicit* identifier can be calculated for some expressions. See "6.7 – Determining Identifiers" for more information.c

The following sections can use the identifier for a root expression, if one is available.

- `orderby`
- `where`
- `having`

Applications that work with the query programmatically will be able to refer to sorting and filtering expressions by name if there is an identifier associated with it.

See "11.5 – Fluent API" for a discussion of where root expression identifiers can be used.

6.7 Determining Identifiers

This section discusses the algorithm used to determine the identifier for a root expression. The intent is to infer an identifier only where it feels intuitive to do so and to require an explicit identifier where inferring one would not be intuitive or would result in a too-generalized identifier (e.g. "First").

In addition, an implicit identifier cannot override an already-existing identifier in the current scope.

6.7.1 Reserved identifiers

The identifiers for all properties and relations of the metaclass of the current scope are assigned first and cannot be overwritten. See "6.7.8 – Conflicts and Overrides" for an example of how to resolve a conflict with a reserved identifier.

6.7.2 Constants

There is no implicit identifier for a manifest constant. The example below shows a few examples where an explicit identifier must be used.

```
Person
{
  FullName:= "{LastName}: {Company.Name}";
  Total:= 45 + 23 * 5;
}
```

6.7.3 Infix Operators

As with constants, an infix operator does not yield an implicit identifier. The example below shows a few examples where an explicit identifier must be used.

```
Person
{
  AmountOrSalary:= Contracts { orderby CreateDate desc }.First.Amount ?? Salary;
  MagicNumber:= Contracts.Amount.Sum + (Salary * 0.3) / 2;
  FormattedDate:= EmploymentDate formattedas 'd';
}
```

6.7.4 Functions

An implicit identifier can be inferred from a function *only if* that function allows an implicit name to be taken from it. Many functions—the aggregation, date and math libraries, for example—cannot be used in this way because the resulting identifier is too generalized (e.g. "First" and "Count" will almost never be specific enough). This decision is made on a function-by-function basis.

The examples below assume that there are two application-specific functions `CompanyID` and `OfficeID` which can be used as the implicit name.

```
Person
{
  CompanyID();                          // Implicit name is 'CompanyID'
  OfficeID(8, 7, 'Home');               // Implicit name is 'OfficeID'
}
```

But none of the following examples yield an implicit name because the math, date and aggregation library functions do not allow it.

```
Person
{
  Round(Contracts.First.Amount, 2);
  (Now - BirthDate).Years;
  TimeEntries {orderby Date }.Last;
  ContractAmount:= Contracts { orderby Amount desc }.First;
}
```

6.7.5 Index Operators

Index operators do not have a name and thus cannot be used as the implicit identifier for an expression. In the examples below, the implicit names `Map` and `RawData` can only be used if they are functions from which an implicit identifier can be obtained.

```
Person
{
  Map['Private'][2];          // Implicit name is 'Map' if function allows it
  RawData['Private', 2];      // Implicit name is 'RawData' if function allows it
}
```

6.7.6 Related objects

A non-trivial expression ending in a property or relation does not have an implicit name. None of the expressions in example below are valid.

```
Person
{
  Contracts { orderby Amount desc }.First.Amount;   // Invalid
  Company { TimeEntries { Project }.First }         // Invalid
  Company { TimeEntries { Count } }                 // Invalid
}
```

To fix the expressions above, assign them to a more meaningful variable name, as shown below.

```
Person
{
  MaxContractAmount:= Contracts { orderby Amount desc }.First.Amount;
  FirstCompanyTimeEntry:= Company { TimeEntries { Project }.First }
  CompanyTimeEntryCount:= Company { TimeEntries { Count } }
}
```

6.7.7 Related lists

As defined in "6.7.1 – Reserved identifiers", identifiers from the metaclass cannot be overridden. Defining an override for scalar properties and objects is straightforward, but is less so for related lists because the contents of a related list can be altered without assigning directly to that identifier.

The distinction is shown in the example below.

```
Person
{
  // This is clearly invalid as a reserved identifier is directly overridden
  Contracts:= Contracts { where Amount > 10_000; orderby Amount desc };

  // Selection and ordering are changed; currently valid, but should it be?
  Contracts { Amount; orderby Amount desc };
```

```
  // Restrictions are changed; currently valid, but should it be?
  Contracts { where Contracts.Amount > 10_000 }
}
```

There are arguments both for and against being able to change the contents of related lists without being forced to assign to a different identifier.

- It can be argued that the identifier "Contracts" in the examples above should always refer to the relation as it was modeled in the metadata. If this is not the case, the consumer of the query results may make incorrect assumptions about the contents of that relation
- On the other hand, the consumer of the query result should be aware of the contents of the query. Forcing a query to use a variable whenever the contents of a related list are modified may introduced unnecessary clutter.

In this current version, an alias is *not* required for related lists which return only part of the modeled contents.

6.7.8 Conflicts and Overrides

If an implicit identifier conflicts with any other identifier in the context, the query must explicitly assign the expression to a variable. A conflict arises when the implicit identifier is the same as an identifier of the metaclass associated with the current scope.

The following example includes an implicit identifier that conflicts with an explicit variable.

```
Person
{
  Total:= 45 + 23 * 5;
  (Total - 45) * 2;           // Implicit name is 'Total'
}
```

The fix is to make turn the implicit variable name into an explicit one to make the intent clear.

```
Person
{
  Total:= 45 + 23 * 5;
  Total:= (Total - 45) * 2;
}
```

This next example includes an implicit identified that conflicts with a property from the metaclass for the current scope.

```
Person
{
  Company.Contact.FirstName  // Implicit name is 'FirstName'
}
```

Simply assigning to a variable name

The only way to resolve this conflict is to assign the result to a variable with a name that does not conflict with any properties in the metaclass for the current scope. In the example below, the explicit variable name is more expressive than the implicit variable would have been.

```
Person
{
  CompanyContactFirstName:= Company.Contact.FirstName
}
```

See the following section "6.8 – Resolving Identifiers" for more information on conflict resolution and overriding.

6.8 Resolving Identifiers

This section describes the algorithm that uniquely determines what each identifier is referring to in a query. Identifiers are either at the root of an expression or part of a dot-notation chain. If an

identifier has parentheses, then it is a function call and is resolved according to the algorithm in "6.8.3 – Matching a Function"; otherwise, it is resolved using one of the algorithms shown below.

6.8.1 Root Expressions

If an identifier appears at the root of an expression, the resolution algorithm checks for a match in the following order:

- Look for a *variable* in the current scope
- Look for a *property* in the current scope
- Look for a *loadgroup* in the current scope
- Look for a *variable* in an outer scope, proceeding from innermost to outermost
- Look for a namespace (e.g. `Date`, `String`, etc.)
- Look for a global *function* (see "6.8.3 – Matching a Function")

6.8.2 Dot-notation Expressions

If the identifier follows a dot-operator, the algorithm changes depending on the identifier before the dot-operator.

- If the identifier is `current` or `predecessor`, the resolution algorithm looks for a *property* of the metaclass in the current scope.
- If the identifier corresponds to a property, the resolution algorithm is:
  - Look for an aggregation *function*
  - Look for an extension *function* for the type of the property (e.g. `Year` if it's a date)
- If the identifier corresponds to a relation with single target cardinality , the resolution algorithm is:
  - Look for a *property* on the target class for the relation
  - Look for an extension *function* for the type of the object
- If the identifier corresponds to a relation with multiple target cardinality , the resolution algorithm is:
  - Look for a *property* on the target class for the relation
  - Look for an aggregation *function*
- If the identifier corresponds to a value (i.e. the result of another function), the resolution algorithm checks for a match in the following order:
  - Look for an aggregation *function*
  - Look for an extension *function* for the type of the value (e.g. `Year` if it's a date)

If the identifier corresponds to a namespace like `Date` or `Text`, the resolution algorithm checks for a match *only* in that namespace.

6.8.3 Matching a Function

When the algorithm looks for a function, it checks for a match in the following order:

- Look for a matching function with the given parameters
- Look for a matching function called as an *extension method*. This means that the first actual argument to the function will be the target of the function call or `current` if the call is not targeted.

The order in which namespaces are searched is not defined. If the same function name exists in multiple namespaces, the only way to guarantee that the right version is called is to qualify the call with the namespace.

The following example assumes that the application has defined a function called `Punchclock.Years`, which conflicts with the `TimeSpan.Years` function in the standard library. The application-specific version cannot be used as an extension method and must be qualified with the namespace.

```
Person
{
  TimeEmployed:= Now - EmploymentDate;
  YearsEmployed:= TimeEmployed.Years;        // Resolves to 'TimeSpan.Years'
  Other:= Punchclock.Years(TimeEmployed)
}
```

6.8.4 Choosing a Function Overload

An actual function call is matched to a formal function declaration in the following manner:

- Any parameters explicitly named in the actual function call must exist in the chosen overload
- Find an overload with the same number of parameters where the actual type of each parameter is the same as the formal type (ignoring default values for parameters)
- Find an overload with the same number of parameters where the actual type of each parameter is the same as the formal type (including default values for parameters)
- Otherwise, the function cannot be called with the given actual parameters
- If a function could be selected, coerce any actual parameters to the formal parameter type, if needed (see "8.13.1 – Type Coercion").

See "11.6 – Function Declarations" for more information on overloads and uniqueness.

6.8.5 Overriding precedence

The following keywords are available to override the default resolution order for root expressions.

Symbol	Usage / Meaning
`property`	A prefix operator that indicates that the identifier that follows it represents a property.
`loadgroup`	A prefix operator that indicates that the identifier that follows it represents a loadgroup.
`global`	A prefix operator that indicates that the identifier that follows it is to be resolved in the global context first (instead of the current context).

The following example illustrates a query that uses the override operators to generate a valid query even when it contains an unfortunately named variable (`Date`).

```
Person
{
  var Date:= Now;
  select
  {
    FirstName; Date; // Variable
    TimeEntries
    {
      orderby property Date;
      where property Date.Year = Date.Year // Variable in outer scope
    }
  }
}
```

6.8.6 The 'property' Override

The following example illustrates a query that returns the first name of the person's company's contact as `FirstName` and the person's first name as `PersonFirstName`.

```
Person
{
  FirstName:= Company.Contact.FirstName
  PersonFirstName:= property FirstName;
}
```

6.8.7 The 'loadgroup' Override

Similarly, the following query includes a loadgroup named `FullName` into the selection, even though there is a variable with the same name already in the selection.[5]

```
Person
{
  FullName:= "{LastName}: {Company.Name}";
  loadgroup FullName;
}
```

6.8.8 The 'global' Override

The following example does not compile because `Date` refers to a property of the metaclass when the intent was to reference the namespace. The identifier `Now` cannot be resolved because `Date` is a property.

```
Contract
{
  default;
  Date.Now
}
```

Use the global keyword to select the namespace instead, as shown below.

```
Contract
{
  default;
  global Date.Now
}
```

[5] This variable is declared using formatting sequences, which are defined in much more details in "8.15.1 – Formatting Sequences".

And, just for completeness, the current metaclass scope is available from the explicit global scope using the `current` keyword. The sample query from section "6.8.1 – Root Expressions" is rewritten below using only fully-qualified references.

```
Person
{
  var Date:= global Date.Now;
  select
  {
    global current.FirstName; Date; // Variable
    global current.TimeEntries
    {
      orderby global current.Date;
      where global current.Date.Year = Date.Year // Variable in outer scope
    }
  }
}
```

6.8.9 Function Call Override

The example below show a query for people that defines a `FullName` property but also calls a function named `FullName`. The query uses parentheses to indicate that it is referencing the function and not the property and then assigns the result to a variable because the implicit identifier of the function is `FullName`, which would *still* cause a conflict.

```
Person
{
  FullName:= "{LastName}: {Company.Name}";
  FullNameFunction:= FullName();
}
```

Now, suppose that a `Person` has a property named `First` (which conflicts with the First aggregation function).

In the following example, the identifier `First` is resolved to the property and the query cannot be evaluated because the identifier `FirstName` cannot be resolved in that context.

```
Company
{
  People.First.FirstName
}
```

In this rather contrived example (collisions between aggregation function names and property identifiers will likely be somewhat rare), the query can resolve the conflict by calling the function explicitly, as shown below.

```
Company
{
  People.First().FirstName
}
```

Alternatively, the namespace could also be used to make the intent clear:

```
Company
{
  Aggregation.First(People).FirstName
}
```

7 Syntax

This section introduces all of the various syntactical elements with an example for each. The meaning and interpretation of the elements are discussed in much greater detail in "8 – Data Types and Operators".

A query comprises one or more expressions, which can be *lines* or *blocks*.

7.1 Lines

A line is a single expression followed by whitespace or a semicolon (see "7.9 – Whitespace" and "7.3 – Separators"). A query can be expressed in a single line or any number of lines can be enclosed in one or more blocks.

The following is a valid query that returns all people.

```
Person
```

However, anything other than a trivial query will contain at least one block, which will then contain more lines.

7.2 Blocks

A block is delimited by curly braces and defines a new context. Blocks can be nested to arbitrary depth. There are two kinds of blocks in a query: *sections* and *relations*. Sections are introduced by one of the reserved keywords (see "7.18.1 – Sections") and relations are introduced by the name of a relation on the class that forms the current context (see "3.3 – Metaclass Scopes").

The following query uses a block to include a restriction that returns only people who work for "Encodo Systems AG".

```
Person
{
  where Company.Name = 'Encodo Systems AG';
}
```

7.3 Separators

Two statements must be separated by a semicolon unless the first statement is a block. The semicolon is optional for the last statement in a block or after a block.

The following is an example with the minimum number of semicolons.

```
Person
{
  FirstName;
  LastName;
  Contracts { select Amount }
  ContactInfo
}
```

Since query texts will also be generated by machines, the following is an example of a query equivalent to the one above.

```
Person
{
  FirstName;
  LastName;
  Contracts { select { Amount; } };
  ContactInfo;
};
```

7.4 Identifiers

Identifiers are case-insensitive and must match the following regular expression:

```
@?[_a-zA-Z][_a-zA-Z0-9]*
```

The leading @-sign is stripped and is included only for queries that need to refer to an identifier from the context that has the same name as a reserved keyword (see "7.18 – Reserved Keywords". The @-sign is always valid even if the identifier is not a keyword (though not very useful).

7.5 Functions

A function call is an identifier with parentheses containing zero or more other expressions as parameters. An example is shown below.

```
Person
{
  where AppFunctions.GetOldest(TimeEntries, 3, 5)
}
```

Applications use functions to integrate custom functionality into QQL. See "8.13 – Functions" for more information.

7.6 Index Operators

Some expressions support an index operator, which has a formal declaration like a function call, but is delimited by square brackets and has no name. Also, an index operator declaration must have at least one parameter. The following query includes several index operators.

```
Person
{
  where
  {
    Contracts[2].Amount > 10_000;
    Map['Private'][2].Enabled;
    App['User1', 3].SecurityLevel > 3;
  }
}
```

Note that `App["User1", 3]` is semantically equivalent to `App["User1"][3]`.

See "8.11 – Index Operators" for more information.

7.7 Dot-notation

Almost any syntactic construct, like *blocks*, *Identifiers*, *functions* and *index operators*, can be chained together using the dot-operator. The following example returns the total amount of all contracts from 2007 for people who started tracking time entries in that year.

```
Person
{
  var YearToCheck:= 2007
  select Contracts { where Date.Year = YearToCheck }.Amount.Sum;
  where TimeEntries.First.Date.Year = YearToCheck
}
```

The semantically valid identifiers to follow any given dot-operator are discussed in far greater detail in "6.8 – Resolving Identifiers".

7.8 Assignment

Assignment is available to assign values or queries to variables than can be used elsewhere in the query. The assignment operator is `:=`. The following query returns all people who work for Encodo Systems AG.

```
Person
{
  var CompanyToSearch:= 'Encodo Systems AG';
  where Company.Name = CompanyToSearch;
}
```

See "6.1 – Variables and Scopes" for more information.

7.9 Whitespace

Whitespace may be freely used to enhance readability, and is only required when separating identifiers from reserved keywords (e.g. `orderby Date` must have a space in order for the parser to recognize that a keyword and identifier were intended).

Imagine that we want to write a query that retrieves from each person their first and last names as well as the list of descriptions for all time-entries from this year with the newest time-entries listed first.

The following query is perfectly valid but is not very legible.

```
Person{FirstName;LastName;TimeEntries{Description;where{Date.Year=Now.Year}orderby
Date desc}}
```

With some whitespace, the intent becomes much clearer.

```
Person
{
  FirstName;
  LastName;
  TimeEntries
  {
    Description;
    where
    {
      Date.Year = Now.Year
    }
    orderby Date desc
  }
}
```

7.10 Comments

Single-line C#-style comments are supported anywhere in the query. There is no multi-line comment construct. See the example below.

```
// Leading comment
Person
{
  FirstName;
  LastName;
  TimeEntries
  {
    // Comment-only line

    Description;
    where
    {
      Date.Year = Now.Year  // End-of-line comment
    }
    orderby Date desc
  }
}
// Trailing comment
```

7.11 Strings

There are different string syntaxes, each of which is appropriate for different situations.

- Double-quote-delimited strings support *formatting sequences*, *formatting groups* and *escape sequences* (see "8.15 – Formatting Text" for more information).
- Single-quote-delimited strings are text-only, but require almost no escape sequences.
- Verbatim strings are prefixed with the @-sign and can contain line feeds to make larger text blocks easier to both edit and read.

7.11.1 Double-quote-delimited Strings

Double-quote-delimited strings are the most versatile strings: formatting groups, formatting sequences and escape sequences are all supported. To include a double quote without terminating the string, it must be escaped (see "8.15.4 – Escape Sequences"). An example is shown below.

```
"This string contains \"double quotes\""
```

A double-quote-delimited verbatim string supports formatting groups, formatting sequences and line feeds but not standard escape sequences. To include a literal double quote, curly brace or angle bracket, use two double quotes, two curly braces or two angle brackets, respectively.

7.11.2 Single-quote-delimited Strings

These are the simplest strings. They can only be a single line and neither formatting nor escape sequences are recognized. To include a single quote without terminating the string, use two single quotes. An example is shown below.

```
'This string contains ''single quotes'''
```

A single-quote-delimited verbatim string is the same as a standard single-quote-delimited string but line feeds are allowed. An example is shown below.

```
@'This is a
verbatim
string with ''single quotes'''
```

An example is shown below.

```
@"This is a
verbatim
string for {FirstName} with ""double-quotes""
and text with <<reserved {{characters}}>>"
```

7.12 Numbers

Queries can include numeric constants as

- Decimal Integers (e.g. 455)
- Floating point decimals in standard notation (e.g. 4.55, 0.55, .55)
- Floating point decimals in exponential notation (e.g. 4.55e3, 4.55e-3, 4.55e+3)

A numeric constant without a decimal point is assumed to be an integer.

7.12.1 Controlling Representation

A number can be followed by one of the following suffixes to coerce the type of the constant:

- "m" coerces to a decimal type, useful for currencies (e.g. 0.55m)
- "f" coerces to a floating-point type (e.g. 0.55f)[6]

If no type is specified, the type of a constant with a decimal point is assumed to be decimal (i.e. "m"). This default was chosen because the data queried by QQL is much more likely to contain currencies than scientific data.

7.12.2 Formatting Large Numbers

To make longer numbers easier to read, the integral part may include underscores. The underscore can be used to separate groups of thousands.

Instead of 1000000001 (one billion and one), it's much easier to read 1_000_000_001.

7.13 Dates, Times and Timespans

Dates, times and timespans can be created with the `Date()`, `Time()` and `Timespan()` functions (see "8.14 – Dates, Times and Timespans" for more information).

7.14 Sets

A constant set is an expression that includes zero or more expressions separated by commas and contained in square brackets. The empty set `[]` and nested sets are supported.

See the "8.9.4 – Set Arithmetic" section to learn how to combine sets.

7.15 Booleans

The keywords `true` and `false` are supported.

7.16 Miscellaneous

The keyword `null` is supported. The SQL concept of an *unknown* value is considered equivalent.

7.17 Reserved Symbols

The valid symbols are listed below. Information on usage and semantic effect can be found in "8 – Data Types and Operators".

7.17.1 Grouping and delimiters

- `{}`
- `()`
- `[]`
- `;`
- `,`
- `:`

[6] No distinction is made between 4- and 8-byte floating-point representations.

7.17.2 Arithmetic

- `+`
- `-`
- `*`
- `/`
- `%`

7.17.3 Comparison

- `=`
- `<>`
- `>`
- `>=`
- `<`
- `>=`
- `!=` (synonym for `<>`)

7.17.4 Miscellaneous

- `??`
- `:=`

7.18 Reserved Keywords

Keywords are case-insensitive. The reserved keywords are listed below. Usage and semantic effect are described in "8.5 – Boolean Operators", "8.7 – Comparison Operators", "8.10 – Text Operators", "4 – Standard Queries" and "5 – Grouping Queries".

7.18.1 Sections

- `var`
- `select`
- `distinct`
- `where`
- `orderby`
- `offset`
- `limit`
- `group`
- `groupby`
- `having`
- `selectObjects`
- `dynamic/include`[7]

[7] Reserved for future use; see "12.3 – Snippets" for more information.

7.18.2 Operators

- `and`
- `or`
- `not`
- `in`
- `isnull`
- `formattedas`
- `matches`
- `contains`
- `beginswith`
- `endswith`
- `matchesregex`
- `cs_` (prefix for comparison operators; see "8.6 – Case-sensitivity Operator")

7.18.3 Resolution

- `loadgroup`
- `property`

7.18.4 Macros

- `default`
- `all`
- `primary`
- `predecessor`
- `current`
- `defaulttext`

7.18.5 Ordering

- `asc`
- `desc`
- `pos`

7.18.6 Filtering

- `empty`

7.18.7 Constants

- `true`
- `false`
- `null`

8 Data Types and Operators

8.1 Types

Each expression has a natural type. Some operators will require that all expressions in the operation have the same type. A query can explicitly set these types or allow the operator to implicitly convert them as needed.

8.1.1 Supported Types

There are only a few base types in QQL.

- Integer
- Float
- Decimal
- Text
- Boolean
- Date
- Time
- Timespan
- Binary
- Object

8.1.2 Implicit Conversion

Each operator and function may define its own implicit conversion rules. The most common case is for comparison operators (see "8.7.1 – Comparing Different Types" for more information) and arithmetic operators (see "8.9.1 – Determining Type" for more information).

8.1.3 Explicit Conversion

In some cases, however, a query will want to treat an expression as a different type (e.g. compare values as text instead of numbers or vice versa).

The following functions are available to convert the type of an expression:

- `Cast(expression, "Type")`
- `Int(expression)`
- `Decimal(expression)`
- `Float(expression)`
- `Date(expression)`
- `Text(expression)` or `String(expression)`
- `Time(expression)`
- `Timespan(expression)`

If the expression cannot be converted to the requested type, the result is null.

8.2 Null-handling

Some expressions will return a null result. A null result can be returned in the situations:

- When a property—like `Person.FirstName`—does not have a value assigned, the result of referencing the property is null.
- When an aggregation operation—like `Contracts.First()`—is applied to an empty collection, the result is null.
- When an aggregation operation—like `Contracts.Amount.Sum()`—includes null values, the result is null.
- When the target of a method call is null—like `Person.Contracts.First.Amount` when the person is null or the contract list is empty—, then any subsequent references are also null.
- When a comparison operator has a null operand—like `Person.ReferenceNumber > 0` when the reference number is either not set or the person is null—the result is always false.
- When an arithmetic operator has a null operand, then the result is null (the concatenation operator, on the other hand, ignores null values).
- When a function returns a null result

Consider the following real-world example that calls the aggregation function `First` to return all people that have at least one contract with an amount greater than 10,000.

```
Person
{
  where Contracts { orderby Amount desc }.First.Amount > 10_000
}
```

If a person has no contracts, then the call to `First` returns a null result and so, as a result, does `Amount`. The comparison to 10,000 returns false because the result of comparing a null value with anything is never true.

The default behavior should suffice in most cases, but there are operators available to fine-tune null-handling (see "8.8 – Null-Testing Operators").

For example, the query above could be modified to use different criteria when there are no contracts available, like *salary*.

```
Person
{
  default;
  MaximumAmount:= Contracts { orderby Amount desc }.First.Amount ?? Salary;
  where MaximumAmount > 10_000
}
```

Note that the query above makes use of the compact notation for *select* statements (see "4.3.4 – Omitting the 'select' Keyword" for more information).

8.3 Operator Binding Strength

There are binding levels which are applied to resolve ambiguities in any expression without forcing the user to add parentheses. In some situations, it aids readability to include these optional parentheses anyway and in others, the parentheses are necessary to return the desired result (see "8.3.1 – Overriding Precedence").

The supported operators are organized into the following binding levels, from weakest to strongest.

Level	Operators
assignment	`:=`
coalesce	`??`
conditional or	`or`
conditional and	`and`
equality	`=`, `<>`, `in`, `matches`, `contains`, `beginswith`, `endswith`, `matchesregex`
relational	`<`, `<=`, `>`, `>=`
additive	`+`, `-`
multiplicative	`*`, `/`, `formattedas`
unary	`-`, `not`, `isnull`, `property`, `loadgroup`

Given these bindings, an expression is unambiguous even without parentheses. The following example returns the sum of all contracts adjusted by a factor of 10% and adds the value of the salary, adjusted by a factor of 20%.

```
Person
{
  Worth:= Contracts.Amount.Sum * 1.1 + Salary * 1.2
}
```

Since multiplication has a stronger binding than addition, the factors will be applied *before* the values are added. The assignment operator has the weakest binding and is thus applied last.

8.3.1 Overriding Precedence

Parentheses can be used to override the default binding strength of operators.

Suppose we want to write a query that returns the maximum amount for all contracts adjusted by a factor of 10% and adds the value of the salary adjusted by a factor of 20%.

The following query relies on the default binding strength for `*` and `+` to yield the expected result.

```
Person
{
  Worth:= Contracts.Amount.Max * 1.1 + Salary * 1.2
}
```

The result is calculated according to the following algorithm:

- `Contracts.Amount.Sum * 1.1` yields `V1`
- `Salary * 1.2` yields `V2`
- `V1 + V2` yields `V3`
- `V3` is assigned to `Worth`

However, the return value of the `Max` aggregation function can be `null` if there are no contracts. In those cases, the query should just use a default value. The following query attempts to fix this problem with the coalesce-operator (`??`) but will yield an *incorrect* result.

```
Person
{
  Worth:= Contracts.Amount.Max * 1.1 ?? 0 + Salary * 1.2
}
```

This query doesn't work as expected because the coalesce-operator has a *weaker* binding than both multiplication and addition. The query above includes the salary multiplied by 20% *only if* the person has no contracts; otherwise, only the sum of the contracts multiplied by 10% is returned.

The calculation is shown below:

- `Contracts.Amount.Sum * 1.1` yields `V1`
- `Salary * 1.2` yields `V2`
- `0 + V2` yields `V3`
- `V1 ?? V3` yields `V4`
- `V4` is assigned to `Worth`

To get the expected result, use parentheses to override the binding strength, as in the example below.

```
Person
{
  Worth:= (Contracts.Amount.Sum * 1.1 ?? 0) + Salary * 1.2
}
```

In this version, the calculation proceeds as expected and as shown below.

- `Contracts.Amount.Sum * 1.1` yields `V1`
- `Salary * 1.2` yields `V2`
- `V1 ?? 0` yields `V3`
- `V3 + V2` yields `V4`
- `V4` is assigned to `Worth`

8.4 Grouping Expressions

Parentheses can also be used to group expressions; this is useful for defining a new expression to be used as the target of a dot-operator.

For example, the following query uses parentheses to extract the number of years from a calculated timespan and return only people who have been employed for ten years or more.

```
Person
{
  where (Now - EmploymentStart).Years >= 10
}
```

8.5 Boolean Operators

The following standard operators are supported for combining Boolean expressions. If necessary, the expressions involved are first converted to Boolean type.

Name	Usage / Meaning
`and`	Returns true if the both the left- and right-hand expressions are true

Name	Usage / Meaning
`or`	Returns true if either the left- or right-hand expression is true
`not`	Returns false if the expression is true and true if the expression is false

8.6 Case-sensitivity Operator

Comparisons applied to expressions of type text are case-insensitive by default. To force case-sensitivity, include the following prefix before the operator.

Symbol	Usage / Meaning
`cs_`	Forces case-sensitive mode for the operator that follows; see "8.7.4 – Case-sensitivity" for examples.

8.7 Comparison Operators

The following standard operators are supported for combining expressions to return a Boolean result. A comparison to null is always false (see "8.2 – Null-handling") and default ordering is described in "8.7.2 – Default Sort Orders".

Symbol	Usage / Meaning
`<`	An infix operator that returns true if the left-hand expression occurs before the right-hand expression in the default ordering for that type
`<=`	An infix operator that returns true if the left-hand expression is equal to the right-hand expression or occurs before it in the default ordering for that type
`>`	An infix operator that returns true if the left-hand expression occurs after the right-hand expression in the default ordering for that type
`>=`	An infix operator that returns true if the left-hand expression is equal to the right-hand expression or occurs after it in the default ordering for that type
`=`	An infix operator that returns true if the left-hand expression is logically equal to the right-hand expression
`<>`	An infix operator that returns true if the left-hand expression is logically not equal to the right-hand expression
`in`	Returns true if all of the elements of the left-hand expression are also elements of the right-hand expression. See "8.7.3 – Evaluating the 'in' Operator"

8.7.1 Comparing Different Types

The comparison operators require that both expressions have the same type. The following rules apply for comparisons:

- If the case-sensitive operator is present, compare as text.
- If the operator is a text operator, compare as text.
- If both sides have the same type, compare as that type.
- If at least one side is an set, the other is converted to an single-element set and the expressions are compared as sets
- If at least one side is numeric and the other can be converted to a number, compare as numbers.
- Compare as text

8.7.2 Default Sort Orders

Operators that return a result based on ordering (<, <=, >, >=) obey the following rules for different types of data.

- *Numbers* use the natural sort order
- *Text* is compared using culture-sensitive sort rules and the current culture of the execution engine.
- Ordering is undefined for all other types (see "11.3 – Ordering Data" for more information)

Comparing to a null value always yields a false result.

8.7.3 Evaluating the 'in' Operator

The `in`-operator is evaluated according to the following rules.

- If either the left- or right-hand side is not a set, it is treated as a set with a single expression.
- The `in`-operator returns true only if each element of the set on the left-hand side is also in the set on the right-hand side.

The following example returns all people where the first name is "John".

```
Person
{
  where 'John' in FirstName
}
```

The query above is equivalent to this one:

```
Person
{
  where ['John'] in [FirstName]
}
```

The following example returns all people where the first, middle or last name is "John".

```
Person
{
  where 'John' in [FirstName, MiddleName, LastName]
}
```

8.7.4 Case-sensitivity

Comparison operators can be combined with the case-sensitivity-operator (`cs_`). The example below finds all people whose last name is "Miller", matching case.

```
Person { where LastName cs_= 'Miller' }
```

The example below finds all people whose last name sorts before "Miller", matching case.

```
Person { where LastName cs_< 'Miller' }
```

The example below finds all people whose last name is either "Miller" or "Baker", matching case.

```
Person { where LastName cs_in ['Miller', 'Baker'] }
```

8.8 Null-Testing Operators

The following standard operators are supported for testing expressions against null.

Symbol	Usage / Meaning
`isnull`	A prefix operator that returns true if the succeeding expression is null.
??	An infix operator that returns the left-hand expression if it is not null; otherwise, the

Symbol	Usage / Meaning
	right-hand expression is returned.

The following example returns all people that do not have any contact information:

```
Person
{
  where isnull ContractInfo
}
```

The `isnull` operator always returns true with sequences; instead use the Empty or Any aggregation function.

The following example returns all people with at least one invoiced contract.

```
Person
{
  where Contracts { where Invoices.Any }.Any
}
```

8.9 Arithmetic Operators

The following standard operators are supported for combining expressions.

Symbol	Usage / Meaning
+	An infix operator that returns the result of adding the right expression to the left expression. See "8.9.1 – Determining Type" for more information about converting input types. If a common non-textual scalar type cannot be determined, concatenate the expressions as text.
-	An infix operator that returns the result of subtracting the right expression from the left expression. See "8.9.1 – Determining Type" for more information about converting input types.
*	An infix operator that returns the result of multiplying the left expression by the right expression. Input expressions are converted according to the rules described in "8.9.2 – Numeric Arithmetic".
/	An infix operator that returns the result of dividing the left expression by the right expression. Input expressions are converted according to the rules described in "8.9.2 – Numeric Arithmetic" and the remainder is included.
%	An infix operator that returns the remainder of dividing the left expression by the right expression. Input expressions are converted according to the rules described in "8.9.2 – Numeric Arithmetic".

8.9.1 Determining Type

The addition and subtraction operators can be applied to two expressions (in its infix form) or to multiple expressions (in its set-operator form). The result of the operation depends on the types of the expressions involved.

- If all expressions are numeric, apply the rules in "8.9.2 – Numeric Arithmetic".
- If all expressions can be added as date/time/timespans, apply the rules in "8.9.3 – Date/Time/Timespan Arithmetic".
- If at least one of the expressions is a set, apply the rules in "8.9.4 – Set Arithmetic".

8.9.2 Numeric Arithmetic

When numbers are combined using any of the arithmetic operators, the resulting numeric representation is determined by the following rules:

- If there is a decimal value, all expressions are converted to decimal and the result is decimal
- If there is a float value, all expressions are converted to float and the result is float
- Otherwise, if the result would include a fractional part, the result is converted to float
- Otherwise, the result is an integer

This example returns a decimal with value `2m`.

```
5m + 2 - .5f * 10
```

This example returns a float with value `2.5f`.

```
5 + 2 - .5f * 9
```

This example returns a float with value `2.5f`.

```
5 + 2 - 9 / 2
```

This example returns an integer with value –`11`.

```
5 + 2 - 2 * 9
```

8.9.3 Date/Time/Timespan Arithmetic

The following operations are supported:

- `Date` +/– `TimeSpan` (result is a `Date`)
- `Time` +/– `Timespan` (result is a `Time`)
- `Date` – `Date` (result is a `TimeSpan`)
- `TimeSpan` +/– `Timespan` (result is a `TimeSpan`)

8.9.4 Set Arithmetic

The following operations are supported:

- `A + B` yields the *union* of `A` and `B`
- `A - B` yields all elements of `A` which are not in `B`

Any expressions in the operation that are not sets are converted to single-element sets.

The following query returns the default properties for each person explicitly excluding two properties and explicitly including another.[8]

```
Person
{
  default - [BirthDate, TimeModified] + Picture
}
```

[8] Since this expression is in the *select* section, the +operator isn't strictly necessary. The query could also be written as follows:

```
Person
{
  default - [BirthDate, TimeModified];
  Picture
}
```

8.10 Text Operators

The following standard text operators are supported. Other string operations can be included with custom functions, some of which are documented in "9.3 – Text".

Name	Usage / Meaning
`formattedas`	Returns a text expression created by applying the format specification in the right-hand expression to the left-hand expression. (See "8.15.2 – Format Strings" for more information.)
`contains`	Returns true if the left-hand expression contains the right-hand expression.
`beginswith`	Returns true if the left-hand expression begins with the right-hand expression.
`endswith`	Returns true if the left-hand expression ends with the right-hand expression.
`like`	Returns true if the left-hand expression matches the "like" expression in the right-hand expression. See "8.10.1 – The `like` Operator" for more information.
`matchesregex`	Returns true if the left-hand expression matches the regular expression in the right-hand expression. See "8.10.2 – The `matchesregex` Operator" for more information.
`matches`	Returns true if the left-hand expression matches the search expression in the right-hand expression. See "8.10.3 – The `matches` Operator" for more information.

8.10.1 The `like` Operator

The right-hand expression supports the following syntax:

- *: matches 0 or more characters
- ?: matches exactly 1 character

Some of the other text operators can actually be emulated using the `like` operator.

Expression	Rewritten with "like"
`FirstName contains "H"`	`FirstName like "*H*"`
`FirstName beginswith "H"`	`FirstName like "H*"`
`FirstName endswith "H"`	`FirstName like "*H"`

8.10.2 The `matchesregex` Operator

The right-hand expression is interpreted as a regular expression. The syntax is based on the .NET regular-expression syntax [3] but only the functionality outlined below is supported.

Character classes

- Positive character groups (e.g. `[a-z]`)
- Negative character groups (e.g. `[^a-e]`)
- `.` – matches any character except for a newline
- `$` – matches the beginning of the input string
- `^` – matches the end of the input string
- `\d` – matches a digit character
- `\D` – matches a non-digit character
- `\w` – matches a word character
- `\W` – matches a non-word character
- `\s` – matches a whitespace character
- `\S` – matches a non-whitespace character
- `\n` – matches a newline
- `\r` – matches a carriage return
- `\t` – matches a tab
- `\` – escapes a reserved character

Grouping constructs

- `()` – grouping expression

Quantifiers

- `*` – matches zero or more times
- `+` – matches one or more times
- `?` – matches zero or one times
- `{n}` – matches exactly *n* times
- `{n,}` – matches at least *n* times
- `{n,m}` – matches at least *n* times, but no more than *m* times

Alternation constructs

- `|` – matches either the left-hand or the right-hand pattern

Examples

The following example finds all people with a first name that starts with "M", ends with "o" and has exactly five characters followed by an optional "s".

```
Person
{
  where FirstName matchesregex 'M[.]{3,3}os?'
}
```

8.10.3 The `matches` Operator

The `matches` operator can be used to perform simple searches against a set of expressions. See "11.2 – Text-matching" for a discussion of performance implications.

The left-hand side is either a single expression or a set of expressions. The right-hand side is a single expression that evaluates to a scalar value. The type of the right-hand side determines which expressions on the left-hand side are included in the operation.

- **Numeric** – properties with numeric and text type are included
- **Text** – all properties with text type are included

If the right-hand side is of type text, then the following syntax is supported:

- Whitespace denotes word boundaries; each individual word must be present in order for the a value to match the expression
- Double-quotes can be used to make a word out of multiple words
- To include a double-quote in the search text, use two double-quotes together

To apply Boolean logic, a query should use multiple calls to the `matches`-operator; see below for examples and "11.2 – Text-matching" and "12.2 – Full-text" for more information on possible future enhancements.

To search for the single word "Encodo Systems" instead of the two words "Encodo" and "Systems", use the following query:

```
Company
{
  Name matches '"Encodo Systems"'
}
```

To find all companies where the default properties have the word "Inc." or "Co" or "AG", use the following query:

```
Company
{
  default matches 'Inc.' or defaulttext matches 'Co' or defaulttext matches 'AG'
}
```

The following query finds all companies where either the name or the description has both "Co" and "Inc." somewhere in the text:

```
Company
{
  [Name, Description] matches 'Inc. Co'
}
```

To find all companies where the name contains "Co" but the word "AG" doesn't appear in the default properties, use the following query:

```
Company
{
  Name matches 'Co' and not default matches 'AG'
}
```

To find all people where at least one of the default properties contains both "Co" and "Inc.", use the following query:

```
Company
{
  default matches 'Co Inc.'
}
```

8.10.4 Case-sensitivity

Comparison operators can be combined with the case-sensitivity-operator (`cs_`). The example below finds all people where the word "Miller" appears in the default search properties, matching case.

```
Person { where 'Miller' cs_matches default }
```

The following example finds all people whose first name begins with "Bo".

```
Person { where FirstName cs_beginswith 'Bo' }
```

8.11 Index Operators

The index operator can be applied to any relation to access the 0-based *n*th element in the relation (if it exists). The following query returns from each person the amount of the third contract for that person.

```
Person
{
  Contracts[2].Amount
}
```

Depending on context, other index operators may be supported, including those with non-integer or even multiple parameters. In the following example, the custom function `Map` returns a dictionary of lists indexed by name; the custom function `RawData` exposes an index operator that takes two parameters.

```
Person
{
  default;
  Map['Private'][2].Amount;
  RawData['Private', 2];
}
```

The index operator is syntactic sugar but is a more natural form for some expressions.

8.12 Miscellaneous Operators and Symbols

The table below lists the non-operator symbols that are valid in a query. A link to the relevant section is included for those operators where there is extended documentation.

Symbol	Usage / Meaning
{}	Delineates the beginning and end of a block (see "7.2 – Blocks")
;	Separates one expression from another (see "7.3 – Separators" and "7.2 – Blocks")
()	Delineates the beginning and end of a list of parameters for a function call (see "8.13 – Functions")
[]	Delineate a set of expressions in a constant set (see "7.14 – Sets"); call an index operator (see "7.6 – Index Operator")
,	Delimits parameters in a function or indexed call (see "8.13 – Functions") or elements in a set (see "7.14 – Sets")
:=	An infix operator that assigns an expression to an identifier; see "6.1 – Variables and Scopes")

8.13 Functions

Functions are not declared anywhere in QQL; rather, they are an implicit part of the execution engine. Some of the standard functions supported by an implementation are documented in "9 – Libraries".

A function is defined by one or more formal signatures, each of which includes *0..n* formal parameters. Formal signatures for a function must differ from one another in either number of parameters or the types and/or positions of parameters.

A function with more than one signature is considered to be *overloaded*. The resolution algorithm matches actual calls against these formal signatures to determine which function to call (see "6.8 – Resolving Identifiers").

8.13.1 Type Coercion

Once a function signature to call has been selected, the types of the actual parameters are coerced to the types of the formal parameters. If the actual type is not correct, the execution engine casts to the formal type. A query can avoid this implicit conversion by providing an actual parameter with the formal type.

8.13.2 Empty Parentheses

Empty parentheses are optional and should be omitted where possible to improve readability. However, if the context contains a metadata identifier and a function with the same name, empty parentheses can be used to choose the function over the identifier. See "6.8 – Resolving Identifiers" for more information.

8.13.3 Optional/default Parameters

A function with *n* parameters may declare that *x* <= *n* of its rightmost parameters are optional. Any formal parameters not supplied by the actual function call are replaced with the default value.

8.13.4 Named Parameters

An actual function call may also assign values to formal parameters by name rather than position. This is useful when calling a function that declares several optional parameters and the actual call only wants to provide a non-default value for one of them. Assume a function declaration with four parameters:

```
F1(percent, slot = 1, attenuate = false, profile = 'high').
```

The following example calls F1 with actual parameters for the first and fourth parameters.

```
F1(23, profile:'low')
```

8.13.5 Execution Efficiency

Some of the basic functions—e.g. text functions (see "9.3 – Text")—will be mapped by the execution engine to the backing store for maximum efficiency, but others will have to be executed locally instead. This has obvious implications for the efficiency of queries using functions. See "11.1 – General Execution" for more information.

8.14 Dates, Times and Timespans

Constant dates must be provided as calendar dates in extended format, as described in the specification for ISO 8601 [2]. Week and ordinal dates are not supported. Constant times must be provided in either local or UTC format.

Some valid examples are shown below.

- `Date('2011-12-10 23:00:00')` yields a *date* value with a *local time*
- `Date('2011-12-10')` yields a *date* value with a *local time* of `00:00:00`
- `Time('23:00:00')` yields a *local time* value
- `Time('23:00:00Z')` yields a *UTC time* value with no offset
- `Time('23:00:00Z+02')` yields a *UTC time* value offset by 2 hours

Constant intervals must be provided in one of the formats described below. The specification for ISO 8601 [2] includes a duration format as well, but it isn't very legible.

The simple format includes slots for days separated by a dot from hours, minutes and seconds, which are given in standard time format. The values for all slots are accumulated and any individual value may exceed the default modulus for that slot (e.g. an interval may include 120 hours even though 24 is the maximum modulus for hours).

Some valid examples are shown below.

- `TimeSpan('9.00:00:00')` yields a value of 9 days
- `TimeSpan('08:30:00')` yields a value of 8 hours, 30 minutes
- `TimeSpan('30:00')` yields a value of 30 minutes
- `TimeSpan('3600')` yields a value of one hour (3600 seconds)

For larger intervals, an extended format is also supported, which includes slots for years, months, weeks, days, hours, minutes and seconds. The format includes one or more number/unit designator groups, like "4 years", "21 weeks" and "72 hours" separated by commas, like "3 years, 2 months".

Some valid examples are shown below.

- `TimeSpan('2 years, 3 months')`
- `TimeSpan('4 hours, 90 minutes')`
- `TimeSpan('5 weeks, 2 days, 12 hours')`

8.15 Formatting Text

The query language offers powerful tools for including formatted text and data in a query result, including formatting sequences and formatting groups, as well as escape sequences and format strings for fine-tuning.

8.15.1 Formatting Sequences

Formatting sequences are useful for building text sequences out of constant text and variables obtained from the data structure.

For example, to return the last name and then the first name of a person separated by a comma and a space, a query can use a formatting sequence as follows:

```
Person
{
  FullName:= "{LastName}, {FirstName}"
}
```

The expression between the curly braces can be more complex as long as it returns a scalar type. This includes function calls that take parameters—even other strings—and related objects and sub-queries. It is, however, recommended to separate formatting from calculation (as shown below).

Though the following is supported, it's not a recommended style; it is included for documentation purposes only.

```
Person
{
  FullName:= "{LastName}: {Contracts \{ where \{ Amount > 500000 \} \}.Count}"
}
```

As you can see, more complex expressions have characters that must be escaped in order to be included and can be confusing. The case above is more elegantly expressed as the following.[9]

```
Person
{
  var
  {
    ContractCount:= Contracts { where { Amount > 500000 } }.Count
  }
  select
  {
    FullName:= "{LastName}: {ContractCount}"
  }
}
```

However, the following expression is still quite easy to read and does not require any local variables or escape sequences, yet it still refers to a property of a related object (the company).

```
Person
{
  FullName:= "{LastName}: {Company.Name}"
}
```

8.15.2 Format Strings

When an expression is included in a formatting sequence, it is converted to a string according to the following rules:

- If metadata can be determined for the expression, the formatting preferences from that metadata is used
- Otherwise, the default format string for that type is used

Naturally, some queries will need to override these default settings, usually for dates or numbers.

[9] The `ContractCount` is declared in the `var` section because the query is only supposed to return the value as part of the formatted `FullName` variable. See "4.2 – Variables" for more information.

The format string is interpreted differently, depending on the type of the expression. Timespans, times, dates, numbers and text all have their own formatting strings, which are defined in the .NET online documentation [1].

Any expression on the right-hand side of the `formattedas` operator is treated as a format string. The example below returns each person's employment date using the standard short formatting (instead of the formatting imposed by the metadata).

```
Person
{
  EmploymentDate formattedas 'd'
}
```

Expressions that appear in formatting sequences also support a shorter version where `formattedas` is replaced by a colon. For example, the following query returns each person's company name and employment date using the standard short formatting (instead of the formatting imposed by the metadata).

```
Person
{
  CompanyDate:= "{Company.Name}: {EmploymentDate:d}"
}
```

8.15.3 Formatting Groups

A formatting sequence includes all constant text in the formatting string regardless of whether it makes sense to do so. For example, the following formatting sequence formats a date and time:

```
TimeEntry
{
  Text:= "{Time} on {Date}"
}
```

If either the `Time` or the `Date` is empty, the value of `Text` will look strange. To avoid this problem, use *formatting groups*, which use an algorithm to determine whether to include a constant string (like "on" in the example above) in the result.

A formatting group includes zero or more elements that are either formatting expressions or constant strings. A constant string is only included in the output text under the following conditions:

- It is the *only* element
- It is the *first* element and the expression *after* it yields a non-empty value
- It is the *last* element and the expression *before* it yields a non-empty value
- The expressions *before* and *after* it yield non-empty values

Formatting groups are introduced by a left angle bracket and closed by a right angle bracket (<>). The example from above can be changed from a formatting sequence to a formatting group by surrounding it with angle brackets.

```
Person
{
  FullName := "<{LastName}, {Firstname}>"
}
```

Now the intervening comma will only be included in the result if both `LastName` and `FirstName` are not empty. The table below shows some sample data and the resulting text.

FirstName	LastName	Result
Thomas	Mann	Mann, Thomas

FirstName	LastName	Result
	Mann	Mann
Thomas		Thomas

Multiple formatting expressions

A slightly more complex example includes a trailing constant and another expression.

```
Person
{
  FullName:= "<{LastName}, {Firstname}: {Company.Name} (Message)>"
}
```

Note that the trailing constant is included whenever *any part* of the expression before it is not empty, not just the `{Company.Name}` formatting sequence immediately before it.

FirstName	LastName	Company.Name	Result
Thomas	Mann	Encodo	Mann, Thomas: Encodo (Message)
Thomas	Mann		Mann, Thomas (Message)
Thomas		Encodo	Thomas: Encodo (Message)
Thomas			Thomas (Message)
	Mann	Encodo	Mann: Encodo (Message)
	Mann		Mann (Message)
		Encodo	Encodo (Message)

Fallback text

The two examples above illustrate that a formatting group returns empty if all of the formatting expressions it contains yield empty values. To avoid empty results, formatting groups can include a fallback text after the coalesce-operator (`??`).

A simple example is shown below.

```
Person
{
  ContractText:= "<Contracts: {ContractDescription}??None>"
}
```

ContractDescription	Result
2 open, 5 closed	Contracts: 2 open, 5 closed
	None

That example is relatively straightforward, but shows how the text can be completely replaced to avoid an empty result.

Nested formatting groups

A more complex example shows that formatting groups can be nested by using the coalesce-operator.

```
Person
{
  FullName:= "<{LastName}, {Firstname}.??<{Company.Name}??Nothing defined!>>"
}
```

This example is no longer quite so easy to read, but it is quite powerful and very concise. The table below shows how the output gracefully adjusts according to the available data.

FirstName	LastName	Company.Name	Result
Thomas	Mann	Encodo	Mann, Thomas.
Thomas	Mann		Mann, Thomas.
Thomas		Encodo	Thomas.
Thomas			Thomas.
	Mann	Encodo	Mann.
	Mann		Mann.
		Encodo	Encodo
			Nothing defined!

And finally, the following example shows a whole sentence that uses formatting expressions, formatting groups and escape characters (which are used to include angle brackets in the final constant text).

```
Person
{
  Legend:=
@"The name is <{LastName}, {FirstName}, {MiddleInitial}.>

The phone number is <{TelNumber}??<{FaxNumber}??\<No contact Info\>>>."
}
```

Determining output for the various inputs is an exercise left up to the reader.

8.15.4 Escape Sequences

Escape sequences are required in cases where a text must include a literal character that either has special meaning (e.g. a double-quote) or which cannot be easily expressed with a standard keyboard (e.g. an m-dash).

The escape character is a backslash. The following standard escape sequences are supported:

- `\t` – Horizontal tab
- `\n` – Newline
- `\r` – Carriage return

```
"This\r\nis\r\na\r\nstring\r\non\r\nseven\r\nlines"
```

Unicode escape sequences are also supported. These take the form of:

- `\u4EA2` – 4-digit sequence
- `\U010A73` – 6-digit sequence

```
"This string contains an m-dash: \u2014"
```

And finally, the following characters can appear after a backslash escape character so that they can be included anywhere in a string without resorting to Unicode.

- \
- "
- <
- >
- {
- }
- ?
- :

So, to include a backslash in a string that supports escape sequences, use a double backslash (\\).

8.15.5 Examples

Assume a query text uses a string that supports formatting groups and formatting expressions. To include a leading angle bracket, use the following escape sequence. Note that the right angle bracket must also be escaped.[10]

```
"This text is in \<angle ?? brackets\>"
```

To include a left curly brace as text, use the following escape sequence.

```
"This text is in \{curly braces\}"
```

To include a left or right curly brace in a formatting sequence, use the following escape sequences. Note that neither the right angle-bracket nor the question mark needs to be escaped in a formatting sequence because those characters have special meaning only within a formatting group.

```
"Open contracts: {Contracts \{ where IsOpen ?? false and Amount > 2000 \}.Count}"
```

Within a formatting group, all special characters must be escaped. The following formatting group includes curly braces around the first name and angle brackets around the fallback expression.

```
"Group: <\{{FirstName}\}??\<EMPTY\>>"
```

The following example is a string that includes an expression that is a call to function `Bar`, which takes a string parameter. This example is included to illustrate that only the double-quote must be escaped; the other characters are not considered special in this context.

```
"Foo: {Bar('{}<>?:\"')}"
```

[10] QQL requires that closing braces and brackets also be escaped both to make parsing the language less context-dependent and also to present a more consistent syntax to the user. It would look somehow unbalanced if the opening bracket was escaped but the closing bracket was not.

9 Libraries

Global functions are declared in a namespace but are also available as global identifiers. The namespace should only be used to resolve a conflict with a metaclass identifier. See "6.8 – Resolving Identifiers" for more information.

Where a function may be called as an extension method, the `this` keyword has been used, as in C#.

Note: This section is subject to change.

9.1 Math

The following functions are available in the `Math` namespace:

- `Round`
- `Abs`
- `Floor`
- `Ceil`
- `IsNaN`
- `IsInf`

9.2 Date

The following functions are available in the `Dates` namespace:

- `Year`
- `Month`
- `Day`
- `Date`
- `Time`

9.3 Text

The following functions are available in the `Text` namespace:

- `SubString`
- `Pos`
- `Format`

9.4 Aggregation

Aggregation functions are applied to a sequence of values. The sequence is defined by the type of expression to which the function is applied. The following list describes the sequences generated by different types of expressions.

- **Scalar**: The sequence for a scalar expression (e.g. `Person.Age`) includes the value returned by evaluating that expression for each row in the result. For example, `Person.Age.Median` returns the median age of all people in the result.
- **Object**: The sequence for an object expression (e.g. `Person.ContactInfo`) includes the objects themselves. Only a handful of aggregation functions—like `Any`, `Empty` or `Count`—are useful for this type of target. Since ordering is not well defined for non-scalar values, functions like `Average` or `Min` are less useful for this target type. Aggregations applied to the metaclass scope (the implicit `current`)

- **Sequences**: The sequence (e.g. `Person.Contracts`) is used directly instead of obtaining its elements from the rows, as with scalar or object targets. Again, since ordering is not well-defined for non-scalar values, functions like Average and Min are less useful, but in addition to those operations supported for object targets, the functions `First` and `Last` are also useful.

The following functions are available in the `Aggregate` namespace.

- `Any` – returns true if the sequence contains element
- `Empty` – returns true if the sequence is empty
- `First` – returns the first element of the sequence
- `Last` – returns the last element of the sequence
- `Min` – returns the minimum value in the sequence
- `Max` – returns the maximum value in the sequence
- `Count` – returns the number of elements in the sequence
- `Sum` – returns the sum of all values in the sequence
- `Average` – returns the average of all values in the sequence
- `Median` – returns the median value of the sequence
- `[]` – returns the element at the given ordinal position in the sequence

9.4.1 Untargeted Aggregations

The following example returns the total number of people whose last name starts with "V".

```
Person
{
  Count;
  where LastName beginswith 'V'
}
```

If an untargeted aggregation operator appears in a grouping section in a grouping query, the target is the group instead of the metaclass. The following example groups people by last name and returns that last name as well as the number of people with that last name.

```
group Person
{
  by LastName;
  select { LastName; Count };
}
```

If the untargeted aggregation operator conflicts with an identifier in the scope, the namespace can be used to disambiguate, as shown below (see "6.8 – Resolving Identifiers" for more information).

```
group Person
{
  by LastName;
  select { LastName; Aggregate.Count };
}
```

9.4.2 Emulating Scalar Results

Aggregation operators can be used to return scalar results.

The following example returns a list of integers (the number of time entries per person):

```
Person { TimeEntries.Count }
```

But this example returns a single integer (the number of time entries for all people):

```
Person { TimeEntries.Count.Count }
```

10 Best Practices

10.1 Defining Variables

Variables can be defined in any section, but the following rules help ensure clarity.

- A variable should only be defined once
- If a variable is used only once, it should be included inline in that section
- Variables that are used more than once should either be defined in the first section in which it occurs or defined in the `local` section
- Variables should be defined in a `select` or `local` section, where possible
- Variables should not be defined in a `distinct` section

10.2 Omitting 'select'

It is recommended to omit the `select` keyword where possible, to improve clarity. The following query returns the first name, last name and the default properties for all contracts for each person.

```
Person
{
  select
  {
    FirstName;
    LastName;
    Contracts
  }
}
```

The query is clearer and shorter when written as:

```
Person
{
  FirstName;
  LastName;
  Contracts
}
```

The `select` keyword is always optional, even when other sections are present.

10.3 Take Advantage of Defaults

Imagine that we want to retrieve all people with default properties[11]. The following formulations are all semantically equivalent.

```
Person
Person {}
Person;
Person { default }
Person { default; }
Person { select default }
Person { select default; }
Person { select { default } }
Person { select { default; } }
```

The first is the shortest and simplest to understand and isn't lacking in expressiveness when compared to the other formulations.

[11] More precisely, we want to retrieve all objects of type *Person*, retrieving for each of the properties in the default loadgroup.

10.4 Remove Clutter

Imagine that we want to retrieve all people with default properties where both the first and last names contain the letter "M". The following formulations are all semantically equivalent.

```
Person { where FirstName contains 'M'; where LastName contains 'M' } }
Person { where { FirstName contains 'M'; LastName contains 'M' } }
Person { where { FirstName contains 'M' and LastName contains 'M' } }
```

The middle one is preferred, however, as is the multi-line style, as shown below.

```
Person
{
  where
  {
    FirstName contains 'M';
    LastName contains 'M'
  }
}
```

10.5 Whitespace and Lines vs. Blocks

The discussion of whitespace (see "7.9 – Whitespace"), introduced the following example.

```
Person
{
  FirstName;
  LastName;
  TimeEntries
  {
    Description;
    where
    {
      Date.Year = Now.Year
    }
    orderby Date desc
  }
}
```

The example above already uses many of the available shortcuts, but it could be reduced even further by eliminating the braces for the `where`-clause, which are superfluous since there is only one expression in the block.

```
Person
{
  FirstName;
  LastName;
  TimeEntries
  {
    Description;
    where Date.Year = Now.Year
    orderby Date desc
  }
}
```

10.6 Dot-notation vs. Blocks

Imagine a query want to return all people that work for a company whose name ends in "AG". A first attempt might yield:

```
People
{
  Company
  {
    where Name endswith 'AG'
  }
}
```

However, note that this query has no default selection and will therefore return the Company relation for each person, but no other fields. The default fields can be restored by including the `default` keyword, but the company would be trickier to remove. Instead, use dot-notation to restrict by the company name:

```
People
{
  where Company.Name endswith 'AG'
}
```

10.7 Consistent Declaration

It is highly recommended to adhere to the following style:[12]

- Each section should appear only once in a given block
- If a section has one expression, use a line; otherwise, enclose multiple expressions in a block.
- The sections in a standard query should appear in the following order: `var`, `select`, `distinct`, `where`, `orderby`, `offset`, `limit`
- The sections in a grouping query should appear in the following order: `var`, `by`, `select`, `where`, `having`, `orderby`, `offset`, `limit`, `selectObjects`

10.7.1 Cleaning Up a Query

The following query is perfectly valid:

```
Person
{
  Contracts;
  limit 10;
  orderby [LastName, FirstName];
  where FirstName contains 'M';
  where LastName contains 'M';
  select FirstName;
  select Lastname;
  orderby BirthDate pos 0;
  where ContractCount < 10;
  offset 2;
  var ContractCount:= Contracts.Count;
  select [ContractCount, MiddleName, BirthDate];
}
```

However, there are a few clarity/consistency issues, listed below.

- The implicit select is used for `Contracts`, but other properties are included with `select` lines.
- The sections are in no clear order
- Variables are used before they are declared; this is technically valid because the `var`-block precedes all other sections in the normalized form of the query
- Many of the sections are needlessly repeated
- The query uses set notation to select multiple properties, misusing the set notation when the {}-notation would be clearer and more standard
- Because the `orderby` declarations are out of order, the `pos` operator must be used to ensure that the query is first sorted by `BirthDate`
- Some properties are selected explicitly and others implicitly; it's easy to overlook the first `Contracts` and think that the `FirstName` is the first field in the selection.

[12] The recommended form differs from the normalized form because the former is intended to be read by humans whereas the latter is intended to be processed by machines. See "6.5 – Normalizing Queries" for more information.

The following is a version of the query above that uses an implicit `select` section, each section appears only once and they are listed in the recommended order. Note that, since `ContractCount` is selected, we don't need to declare it in a variable section.

```
Person
{
  Contracts;
  FirstName;
  LastName;
  ContractCount:= Contracts.Count;
  MiddleName;
  BirthDate;
  where
  {
    FirstName contains 'M';
    LastName contains 'M';
    ContractCount < 10
  }
  orderby
  {
    BirthDate;
    LastName;
    FirstName
  }
  offset 2;
  limit 10
}
```

There is only one small and somewhat subjective change that we can make to improve readability and maintainability. If a variable is used *only* in the select, it should be declared directly there; otherwise, it is better to use an explicit select so that all calculations are in one place. The variable `ContractCount` barely qualifies as a calculation, but it is used in the `where`-section as well and should be extracted.

Therefore, the final, recommended format for the query is shown below.

```
Person
{
  var
  {
    ContractCount:= Contracts.Count;
  }
  select
  {
    Contracts;
    FirstName;
    LastName;
    ContractCount;
    MiddleName;
    BirthDate
  }
  where
  {
    FirstName contains 'M';
    LastName contains 'M';
    ContractCount < 10
  }
  orderby
  {
    BirthDate;
    LastName;
    FirstName
  }
  offset 2;
  limit 10
}
```

A more compact form that makes better use of horizontal space is also acceptable (although it makes it more difficult to add/remove/comment individual expressions).

```
Person
{
  var
  {
    ContractCount:= Contracts.Count
  }
  select
  {
    Contracts; FirstName; LastName; ContractCount; MiddleName; BirthDate
  }
  where
  {
    FirstName contains 'M';
    LastName contains 'M';
    ContractCount < 10
  }
  orderby
  {
    BirthDate; LastName; FirstName
  }
  offset 2; limit 10
}
```

10.8 Common Pitfalls

10.8.1 Losing the Default Selection

Consider the following query, which implicitly selects the default properties for a person:

```
Person
{
  where Contracts { orderby Amount desc }.First.Amount > 10_000
}
```

A naive refactoring to return the value to which 10,000 is compared would look like this:

```
Person
{
  MaximumAmount:= Contracts { orderby Amount desc }.First.Amount;
  where MaximumAmount > 10_000
}
```

However, now the default properties for the person are no longer returned because an explicit identifier, `MaximumAmount`, was added to the selection. To include the default properties again without explicitly listing them all, use the `default` keyword, as shown below.

```
Person
{
  default;
  var MaximumAmount:= Contracts { orderby Amount desc }.First.Amount;
  where MaximumAmount > 10_000
}
```

10.8.2 Selecting instead of Ordering

The following query looks like it returns all people sorted by `BirthDate`, then `LastName`, then `FirstName`.

```
Person
{
  orderby BirthDate; LastName; FirstName
}
```

However, it actually returns the `LastName` and `FirstName` for all people, sorted by `BirthDate`. In normalized form, the query above is actually:

```
Person
{
  select LastName;
  select FirstName;
  orderby BirthDate
}
```

Since the semicolon is a line terminator, the `FirstName` and `LastName` are interpreted as expressions in the implicit `select` section instead. To get the desired result, use a block for the `orderby` section.

```
Person
{
  orderby { BirthDate; LastName; FirstName }
}
```

10.8.3 Aggregating Relations in Grouping Queries

The following query looks like it returns the `LastName` of the people in each group as well as the `BirthDate` and the total amount of time for the oldest member.

```
group Person
{
  by LastName;
  select
  {
    LastName;
    BirthDate.First;
    TimeEntries.Amount.First.Sum
  }
}
```

However, it actually returns the sum of the amounts of the first[13] time-entry for each person in the group. To get the desired result, switch the two aggregation operators.

```
group Person
{
  by LastName;
  select
  {
    LastName;
    BirthDate.First;
    TimeEntries.Amount.Sum.First
  }
}
```

[13] That is, the first entry is taken from the list of time-entries obtained by sorting by the default ordering in the metadata (if one is available).

11 Implementation Details

11.1 General Execution

It is up to any implementation to do the following:

- return those results quickly and accurately
- return an error message indicating to the user which portions of the query could not be applied
- return those results along with a list of hints and warnings indicating where the query formulation has potential performance problems

If a backend to which the query is mapped (e.g. an SQL database) does not support the full expressiveness of the Quino query language, the execution engine should make an attempt to fill the gaps with client-side evaluation, if possible. It is understood that there are some classes of query that require a large amount of data to be transferred in order to perform client-side evaluation (e.g. When a query is ordered or grouped by an expression that cannot be mapped). In those cases, the engine should intelligently determine what the likely performance cost is and choose between aborting execution and issuing a warning.

While it is up to the execution engine to return the expected results as efficiently as possible, the application will also have to make some concessions for the sake of performance.

11.2 Text-matching

It is understood that a full implementation of the `matches`-operator will lie outside the scope of an initial or simple execution engine, especially if it is to make use of high-performance text searching (commonly called "full text" search).

However, a cheaper implementation that satisfies many application requirements is possible by converting to expressions that use the `contains`-operator and `like`-operator instead.

Consider the first example from "8.10.1 – The `matches` Operator", repeated below.

```
Person { where 'Miller' matches default }
```

Let us assume that `default` refers to the following properties: `FirstName`, `LastName`, `Initials` and `Description`. In that case, the query above is semantically equivalent to:

```
Person
{
  where
  {
    FirstName contains 'Miller' or
    LastName contains 'Miller' or
    Initials contains 'Miller' or
    Description contains 'Miller'
  }
}
```

To avoid matching the `Initials` and `Description`, an application might use the following query:

```
Person { where 'Miller' matches default - [ Initials, Description ] }
```

This would, in turn map to the following query:

```
Person
{
  where
  {
    FirstName contains 'Miller' or
    LastName contains 'Miller'
  }
}
```

It is clear that this mapping does not scale for large data sets but it is a perfectly adequate implementation for smaller data sets.

It is recommended to use the `matches`-operator because an execution engine that supports full-text searching can more easily optimize that operator than calls to the `contains`-operator.

11.3 Ordering Data

Basic types—like text, dates and numbers—have a natural ordering. Compound types—like `Person` and `Company`—are naturally ordered by their primary key, but this isn't usually a very semantically useful ordering. However, the execution engine is free to provide orderings for other types.

The ordering for expressions that correspond to objects is technically undefined but could make use of the default ordering defined in the metadata to compare the two objects. Naturally, this presupposes that both objects are based on equivalent metadata. However, even if an execution engine provides this functionality, the ordering is still technically undefined as far as the standard is concerned.

11.3.1 Database-dependent Sorting

Comparisons and ordering will be mapped to a database backend to improve performance. However, this means that the ordering may be applied differently than described in "8.7.2 – Default Sort Orders". In particular, the text will be sorted according to the collation determined by the execution engine (which can be that of the database or the connection or even that defined on the user's local machine).

11.4 Escape-sequences

In double-quoted strings, not only the opening, but also the closing braces and brackets of formatting groups and sequences must be escaped. One could argue that it suffices to escape the opening brace or bracket. If there is never an opening brace or bracket detected, then the parser need not be told to ignore the closing brace or bracket.

However, this would require that a parser be more context-aware and dependent. It would look somehow unbalanced if the opening bracket was escaped but the closing bracket was not, as shown in the example below:

```
@"This is not a formatting group: <<{{FirstName}>"
```

Compare to the actual format:

```
@"This is not a formatting group: <<{{FirstName}}>>"
```

11.5 Fluent API

It's also possible to design an API to create queries in code instead of by parsing query texts. An implementation will likely define an in-memory representation of the query that can be created by a parser or by calling methods in a fluent API.

The in-memory representation should provide an API to allow an application to retrieve the following information from a query:

- The root scope
- Any nested scopes in a given scope
- The sections of a scope (e.g. `select`, `where`, `orderby`, etc.)
- The root expressions of a section, accessible in declaration order
- The root expressions of a section, accessible by name, if available (see "6.6 – Root Expressions & Identifiers" and "6.7 – Determining Identifiers" for more information)
- All expressions in a query

11.6 Function Declarations

Implicit in the algorithm described in "6.8.4 – Choosing a Function Overload" is the notion that each overload of a function must have a unique signature *for the given context*. It is possible for the query to resolve the conflict by including a namespace. It is up to the evaluation engine to enforce uniqueness within namespaces.

A few examples should help. Assume that the application has registered two namespaces—`A` and `B`—with the evaluation engine, each with the function `f(int)`. This is a perfectly legitimate configuration, but the following query cannot be resolved unambiguously.

```
Person
{
  where f(Contracts.Count)
}
```

In this case, the query must specify the namespace, as shown below.

```
Person
{
  where A.f(Contracts.Count)
}
```

However, imagine that the namespace `A` includes the following function declarations:

```
f(int paramOne, int paramTwo=0)
f(int paramOne, int paramTwo)
```

Because the first version may be called as `f(2)` whereas the second cannot, these look like overloads. However, the resolution rules consider these two overloads to be equivalent[14] so the execution engine must prevent such a situation from occurring by validating the namespaces and functions that are added to it.

[14] As does C# or pretty much any statically typed language with method overloads.

12 Future Enhancements

12.1 Parameters

QQL could allow text to include external variables that are supplied when the query is executed. One possible syntax is to $-notation to indicate that a variable is a query parameter.

The example below returns all people whose biggest contract is greater than an external value named `$minContractSize`.

```
Person
{
  where Contracts { orderby Amount desc }.First.Amount > $minContractSize
}
```

12.2 Full-text

There is currently no explicit support for full-text searching, although the lightweight `matches`-operator is intended to fill this gap to some degree. There are a few ways to extend support in this area:

- Extend the syntax supported by the `matches`-operator to include Boolean operations, scoping with parentheses and possibly proximity or boosting operators.
- There is currently no standard query language for full-text searches. An application may add a function to accept an implementation-specific query and extend the execution engine to pass this text to the back end for mapping.
- map full text integrate support for a specific solution (e.g. *Lucene*)

12.3 Snippets

QQL could also allow texts to be written that do not refer to a single model or metaclass context. Instead, the queries could be included in other queries and inherit the context from there. These snippets of query text would be introduced by the keyword `dynamic` included in other queries with the keyword `include`.

Dynamic queries could also declare parameters, as shown below.

```
dynamic RestrictHistory(minHistorySize)
{
  where History { orderby Date desc }.First.Size > $minHistorySize
}
```

The snippet above could be included in another query as follows:

```
Person
{
  where Contracts { orderby Amount desc }.First.Amount > 10_000;
  include RestrictHistory(3)
}
```

In normalized form, this query looks like:

```
Person
{
  where Contracts { orderby Amount desc }.First.Amount > 10_000;
  where History { orderby Date desc }.First.Size > 3
}
```

12.4 Ad-hoc Relations

Currently, the only way to include sub-object results in a query result is by referring to a relation in the current metaclass scope (e.g. `TimeEntries` when the scope is `Person`). However, it

would be interesting to be able to include the results of other queries as sub-objects as well, even if there is no relation defined in the model.

The following query returns all people, each with a list of customers where the contact person has the same last name.

```
Person
{
  Nepotists:= class Customer where Contact.LastName = LastName
}
```

The second reference to `LastName` is to the property in the `Person`; however, if the `Customer` had this property as well, it would have to be referenced explicitly using the `predecessor` keyword:

```
Person
{
  Nepotists:= class Customer where Contact.LastName = predecessor.LastName
}
```

12.5 Cross-model Ad-hoc Relations

Once ad-hoc relations are available, it's a short step to including data from other models in a query result, as shown in the query below, which selects customer data from a model named `Base`. The model name before `Person` is optional, but is included for clarity.

```
Punchclock.Person
{
  Nepotists:= class Base.Customer where Contact.LastName = LastName
}
```