

FQL - Federated Query Language

A query language for structured and distributed data.

1. Purpose

FQL is a query language, that allows retrieval of data from diverse data sources in a unified manner.

It is based on the syntax of SQL, but it has a different execution and data model. The data model and the available operations are richer, allowing better modelling of the problem domain, and easier querying of the stored data.

It defines a core language and specifies an execution environment to run the queries and to return the resulting data.

FQL can be used to retrieve data:

- for data manipulation in an application,
- reporting and business intelligence.

The specification of FQL is independent of the the database system, the operating system, the physical data layout and the implementation language.

FQL is flexible enough to query data from different sources e.g. relational databases, object-oriented databases, graph databases, document databases, key/value stores etc.

The execution model permits accessing data from different databases and database systems in one query, thus providing a federated view of the data.

It is also intended to serve as a native query language for newly implemented database systems, that wish to provide a dynamic query facility for their systems.

FQL is purely a retrieval language. It does not specify a "Data Definition Language" (DDL) nor a "Data Manipulation Language" (DML). These tasks are left to the native database APIs.

2. Overview

FQL is functional language. Its elements (clauses) are operators that transform the data source into the result data.

It's behaviour is described by the functions, that the operators perform on the data.

Like SQL it is not a procedural language, nor is it usable as a general programming language. (It is not computationally complete)

Data is passed from one operator to the next by means of *iterables*. An Iterable is a data container that can access its elements one by one. Examples of iterables are relational tables, arrays in a programming language, or iterators. An important aspect of an iterable is that it may have a physical ordering. Items from an iterable are processed in this order, unless this is changed by an 'order by' clause. If a data source (e.g. a relational table) does not maintain a physical ordering, then the query can not take advantage of this order, e.g. by using the operator 'previous'. Examples of iterables with a physical ordering are arrays, lists, or the tags of an XML document.

Execution happens, when the resulting iterable is advanced to the next item. It will then call its base iterator to deliver data, until it has enough data, so that an output item can be formed and returned as

the next item.

To optimize performance an FQL implementation may choose to cache intermediate results, to merge several clauses into one, or choose different execution plans. However it must ensure, that data coming from an ordered source (array, list) maintains its natural ordering.

3. A first example

```
from orders
where date > parseDate("10-06-29")
      and customer.address.country = 'D'
      and count items > 2
select order_id, customer, date, items
```

This looks very familiar to users of SQL and even more familiar to users of Microsoft LINQ. In fact at this basic level it has a lot in common with a LINQ query.

But one must be aware, that the interpretation is quite different from SQL and uses features, that are not available in SQL.

How is this query to be interpreted? Let us examine the operations row by row:

1. from orders

This creates an initial iterator, by taking a element out of the initial scope that is provided by the data source. The elements of "orders" are not just the rows of a relational table. They are proper objects. I.e. they can contain other objects, or they can contain collections of objects.

The initial iterator gives access to all elements and all fields of the entry point.

2. where date > parseDate("10-6-29")

This takes the original iterator and applies the filter expression to it returning an iterator, that only yields those elements, which match the filter condition. The string is parsed into a date for comparison.

3. and customer.address.country = 'D'

This uses a navigation path to access the customer field of the order (which is a nested or referenced object) and to navigate to the address field and from there to the country. The dot is a generic operator, that relies on the specific database driver to retrieve the related object. This may happen through a relational join, a lookup in a key/value store, by following a link in a graph database or by accessing a nested object in a document database. However for the logic of the query the access method is not relevant. The independence of the underlying access method makes FQL so simple and powerful.

4. and count items > 2

This accesses an embedded collection (the order items) and applies an aggregation operation to it which is then used in the filter. FQL is designed to make handling of complex nested data structures easy.

5. select order_id = id, Customer, order_date, items

This takes the order values and upon each iteration returns a compound object (document), that contains the four fields mentioned. One must be aware, that the fields are not required to be atomic. "Customer.address" is most likely a compound object, that is accessed through a reference. And the "Items" can be a (and most likely are) a collection of the order items.

What can be learnt from this simple example?

- FQL looks like SQL but the order of select, from, where is exchanged.
- Attributes can be multivalued.
- The semantics are clearly defined by the stepwise interpretation of the clauses.
- Any serious implementation should be able to merge the three operations into one, if this results in better performance.

4. Understanding Scopes in FQL Queries

To understand an FQL Query one must be aware of the associated scope.

A scope is a set of names, that can be used at some point in a query. The scope is modified or replaced by each clause in the query.

The scope contains all elements that are directly addressable. I.e. that have a name, that can be used in an expression. At the end of a clause a new scope is created, which defines the names that can be used in the next clause.

Initially the scope is empty. A 'connect' clause opens a database and adds the entry points of the database to the scope. (In a relational database setting, the entry points would be the tables)

A from clause takes one element from the scope and replaces the scope with the names, that are directly accessible in this iterable. These would be the fields of the table.

A select clause creates a new scope, that only contains those names, that are listed in the clause.

5. Federation

Federation means, the combination of data from several independent data sources.

I.e. the data comes not only from one database or system, but from a number of 'subsystems' that can operate independently. I.e. each subsystem is a fully functional database system, that could operate on its own.

To achieve this the query processor needs the ability to access other (remote) databases, apart from the one it currently runs on.

It is also possible to operate the query processor independently from all subsystems. It then treats all its databases as "remote".

Therefore the query processor has a two layered architecture. The lower level, which gathers the data from the subsystems, and the upper layer, which combines the data from the subsystems.

Since the subsystems are independent, they each have their own native query mechanism. The mapping from FQL to the native queries happens through the use of driver modules, that can be plugged into an FQL query processor. The upper layer and the native driver plugin cooperate to optimize and plan the execution of a query. The API between the upper layer and the driver plugins is part of the FQL specification.

6. Introduction to the query clauses

This chapter gives an overview of the query clauses used in FQL. It is not a complete and formal specification, but notes the most important features of each clause. The syntax of each clause is specified using BNF. With optional parts enclosed in brackets '[]', repeated (0 or many repetitions) elements are enclosed in braces '{}', alternatives are separated by the bar character '|' and keywords are bold and enclosed in quotes.

6.1 From

The from clause starts a query. It has one operand which selects an iterable from the scope.

Note: This is different from SQL which allows several tables to be listed in the from clause.

```
from_clause ::= 'from' [name '='] identifier
```

6.2 Where

The where clause is an expression, that is applied to each element in the current iterable. If the expression returns true, the item is passed on, if it evaluates to false, then it is skipped.

```
where_clause ::= 'where' expression
```

6.3 Select

The select clause constructs a new item from the fields of the item that is being iterated over. The select statement consists of a comma separated list of expression, that compute the fields of the new item. A select clause can construct items, where a field is another multivalued item. (nested object) or where a field is an array of items (nested collection). A select statement may contain nested queries, which have to be enclosed in parentheses.

A select clause may specify the keyword *ref*, which causes the query processor to return references (implemented as keys or oids) to the selected items instead of the items themselves. The items remain in the database, and can be retrieved later using the references.

```
select_clause ::= 'select' ['distinct'] ['ref'] expression_list
```

6.4 Join

The join clause combines data from two iterables into one based on the equality of two expressions. (Equijoin)

The join clause may specify the keyword *group* which causes the query to return a grouped structure. I.e. for each object on the left, an iterable of the matching elements on the right is returned. The inner iterable is returned as a collection member of the left item.

```
join_clause ::= [['left' | 'right'] 'outer'] ['group'] 'join'  
[ name '=' ] iterable 'on' expression '=' expression
```

The join operation returns an iterable, that contains the fields from both sources. It behaves as in SQL with the exception of the group join. The group join returns the items from second source grouped into iterables that are embedded as fields in the first iterables items. It creates an iterable of source one items, which all contain an iterable for the corresponding items from iterable two. It behaves like a join operation followed by a group operation.

6.5 Group

A grouping creates an iterable that enumerates the groups found in the source iterable.

Each group contains the key of the group and all other fields that are specified in the grouping statement. Fields in a group can be aggregates, formulas based on aggregates and a list of the values (members) that belong to the group.

The list of the groups values may be a nested grouping. A nested grouping is introduced with the keyword 'by'.

Aggregations are created with the aggregation operators: sum, min, max, avg, first, last, count.

All fields may be assigned to chosen names by writing: name '=' expression.

The default names for the group key is 'key', for the groups values it is 'values'. The default names of aggregations only exists for simple aggregations. (I.e. aggregations of a single field. For these

simple aggregations the default name is the name of the aggregation operator followed by an underscore character ('_') and the name of the field.

Formulas may be expressed based on the key (and its dependent fields) and aggregations. Formulas have no default name, so a name must always be chosen.

The fields named in the group clause form the scope for subsequent clauses.

A where clause after the group clause references the scope created by the group clause. It behaves like the 'having' clause in SQL.

The aggregations are operators in FQL and not functions. Thus it is possible to omit the parentheses.

Aggregations may reference fields from the source scope or aggregations or formulas in nested groups.

```
group_clause ::= 'group' 'by' assignment ',' {inner_grouping ','}
expression_list
```

```
inner_grouping ::= [name ':='] by '(' expression_list ')'
```

6.6 Flatten

The flatten clause is the inverse operation of the group or group-join clause. It iterates over a nested collection and returns the inner element together with the outer element.

```
flatten_clause ::= 'flatten' expression
```

6.7 Enumerate

The enumeration clause servers to recursively process a graph structure. The expression list specifies, which pathes shall be evaluated recursively. I.e. For each element in the source iterable, the recursion expressions are evaluated. If they return a null value, then each value is added to the result iterable and the recursion expressions are applied to them. Recursion stops if an expression returns null, or if a data item has been encountered before. To limit the depth of the recursion, the built in function depth can be used. It returns the distance of an item from the root of the recursion.

```
enumerate_clause ::= 'enumerate' {expression ','} expression
```

6.8 Concatenation

The concatenate clause combines the results of two separate queries into one long iterable.

```
concatenate_clause ::= 'concatenate' query
```

6.9 Get

The get clause is used for direct indexed access. It is used in situations where the index is not part of the item and can not be specified as part of the where clause. This can be the case in key/value based data stores or in object databases.

It can also be used to limit the number of objects in a query to a certain range.

The '<<' operator serves to specify an index range that excludes the upper bound.

```
get_clause ::= 'get' { index_set ',' } index_set
index_set ::= expression '..' expression
            | expression '<<' expression
            | expression
```

6.10 Ordering

The resulting iterable can be sorted.

```
order_clause ::= 'order' ['by'] { ordering_key ',' } ordering_key
```

```
ordering_key ::= expression [ 'ascending' | 'descending' ]
```

7. Expressions

Arithmetic or computational expressions appear in several places in a FQL query. Their most important use is as the condition in the where clause, and for constructing values in the select clause.

Expressions have a type, which is either a boolean, a number, a string, an array or an object. Arrays are created by enclosing them in brackets '[]' and objects by enclosing them in braces '{}'.

8. Navigation

Navigation using the dot notation is a very important feature of FQL. E.g. one can write:

```
some_order_item.order.customer.address.zip
```

Using navigation pathes makes a query readable and database access efficient.

An array can be accessed by using the index notation. E.g. we can write:

```
some_teacher.classe[1]
```

which returns the first item of the list of classes that a teacher teaches.

9. Preliminary clauses

FWL offers a number of clauses, that simplify the management of complex queries, and that can be used to define a database independent layer.

9.1 Named queries

A define clause gives a name to a query. This name becomes part of the initial scope, and can be used later in a from clause.

```
named_query ::= 'define' name query
```

9.2 Import clauses

An import clause can be used to load a separately defined set of named queries. The named clauses in an imported query file can be used to create a layer on top of the physical data entry points. E.g. a named group join can preprocess a relational structure into an object graph, that is easier to query than a flat view. Or a object graph can be turned into a flat table using the flatten operator. Thus the queries of the imported file shield the dependent queries from changes in the underlying structure.

```
import_clause ::= 'import' {url','} url
```

9.3 Database Open

An FQL query can specify the parameters that are required to open a database. The parameters are passed to the database driver to open the database. The entry points of the opened database are added to the initial scope.

10. Examples

10.1 Grouping

```
from orders
group
  by date.yearmonth,
     countries = (by country
```

```
    minprice = min price,  
    maxprice = max price,  
    price_range = maxprice - minprice;  
    select orders  
  ),  
  sum price,  
  min pricerange,  
  max pricerange,  
  price_range_from = min max price  
  price_range_to = max min price  
where price_range > 100
```

This example created a nested grouping and calculates a number of aggregates. It shows advanced usage of the aggregation operators and of intermediate results.