

μ QL: the Muon Query Language

Fred Gray
University of California, Berkeley
DRAFT

August 1, 2003

Historically, the `mu` analysis program had a mechanism known as `superq` that found coincidences between the TDC and other detector elements. It also had the ability to produce an `HBOOK` column-wise `ntuple` from the output of `superq`. However, the code implementing `superq` grew to be untenable, and it was removed. This document describes the replacement for `superq`, a new mechanism for generating cross-detector coincidences. The Muon Query Language (μ QL) is a special-purpose high-level language with a relatively simple syntax in which coincidences, histograms, and `ntuples` may be specified compactly and conveniently. It is processed by a translator that generates C++ code suitable for incorporation into the MIDAS analyzer framework. Consequently, it permits the user to do a great deal of μ Cap analysis work without detailed knowledge of C++.

1 Introduction

In 1969, E.F. Codd proposed the *relational model* that has become the basis of essentially all modern database systems. It provides a mathematical foundation with which to reason about database queries. Data are arranged in tables (formally known as *relations*) such as the following:

Table <i>Owners</i>			Table <i>Dogs</i>		
Name	Address	Phone number	Owner	Dog	Breed
Wilma	1100 Hartford Avenue	523-1546	Wilma	Vinnie	Border collie
Milo	2419 California Street	721-0559	John	Fluffy	Toy poodle
John	1940 Tyvola Road	537-3646	Milo	Ursula	Beagle
			Wilma	Outback	Australian shepherd

These two tables might be present in a database belonging to a pet store, for instance. The tables consist of columns (formally, *attributes*) identified by names, and rows (). There is no natural ordering within the table, either for the columns or the rows; they are treated as unordered sets. Moreover, each row is required to be distinct from all other rows in the same table. Each entry in the table is single-valued; information is organized into a normalized form so that there is

no need for array-valued cells. In the example, it is seen that owners (notably Wilma) may have multiple dogs, so the relation from owner to dog is given in its own table.

Codd defined a *relational algebra* that includes a number of operations on tables. One of the more interesting operators is the Cartesian product $A \times B$, which yields a new table in which each row from A is joined with each row from B . In the example, $Owners \times Dogs$ would be

Table $Owners \times Dogs$

Name	Address	Phone number	Owner	Dog	Breed
Wilma	1100 Hartford Avenue	523-1546	Wilma	Vinnie	Border collie
Wilma	1100 Hartford Avenue	523-1546	John	Fluffy	Toy poodle
Wilma	1100 Hartford Avenue	523-1546	Milo	Ursula	Beagle
Wilma	1100 Hartford Avenue	523-1546	Wilma	Outback	Australian shepherd
Milo	2419 California Street	721-0559	Wilma	Vinnie	Border collie
Milo	2419 California Street	721-0559	John	Fluffy	Toy poodle
Milo	2419 California Street	721-0559	Milo	Ursula	Beagle
Milo	2419 California Street	721-0559	Wilma	Outback	Australian shepherd
John	1940 Tyvola Road	537-3646	Wilma	Vinnie	Border collie
John	1940 Tyvola Road	537-3646	John	Fluffy	Toy poodle
John	1940 Tyvola Road	537-3646	Milo	Ursula	Beagle
John	1940 Tyvola Road	537-3646	Wilma	Outback	Australian shepherd

At first glance, this operation does not appear to have been very useful. However, it is a useful step towards the definition of an operation called the θ -join $A \theta(A, B) B$. This operation takes the Cartesian product of two tables and restricts it to those rows satisfying some condition. In the example, it would be natural to take the condition $Owners.Name == Dog.Owner$, effectively yielding the set of $(Owner, Dog)$ pairs:

Table $Owners \theta(Owners.Name == Dog.Owner) Dogs$

Name	Address	Phone number	Owner	Dog	Breed
Wilma	1100 Hartford Avenue	523-1546	Wilma	Vinnie	Border collie
Wilma	1100 Hartford Avenue	523-1546	Wilma	Outback	Australian shepherd
Milo	2419 California Street	721-0559	Milo	Ursula	Beagle
John	1940 Tyvola Road	537-3646	John	Fluffy	Toy poodle

Indeed, this operation is so natural that it would even be called the *natural join* of the two tables: a θ -join between the *primary key* of one table and its representation as a *foreign key* in the other.

Codd proceeded to develop a *relational calculus*; the word *calculus* is used to mean “a system for doing calculations” and in this case has nothing to do with derivatives or integrals. He proved that it was equivalent in power to the algebraic formalism, and he proposed a query language called ALPHA based on it. Today, nearly all database systems are based on a query language called SQL (structured query language) that is loosely derived from ALPHA. In SQL, the simple query “Where does Wilma live?” would be written

```
SELECT Address FROM Owners WHERE Owners.Name='Wilma'
```

The more complex θ -join example above would be written as

```
SELECT * FROM Owners, Dogs WHERE Owners.Name=Dogs.Owner
```

The asterisk means that all columns from both tables should be included in the result.

The author submits that the analysis of data from experiments such as μ Cap (and also μ Lan) involves a substantial amount of processing that can conveniently be written in terms of a quasi-relational model. Data from various detector systems arrive in parallel streams that may be regarded as tables:

Table <i>HITS</i>		Table <i>TPC</i>		
Time	Parameter number	Entrance time	Endpoint time	Endpoint anode
1550	6000	1557	3722	39
1553	6002	5917	9002	41
4561	7011	:	:	:
4566	7031	:	:	:
:	:	:	:	:

The formation of coincidences between these detectors is a fundamental step in the data analysis. Code written manually to construct these coincidences tends to grow out of hand because there are so many different combinations to be tried. At some level, all of this code ends up looking the same, often with the same mistakes tediously made and corrected in each instance. It will be useful to have a tool that generates this code from a more compact high-level description. This tool will also serve as a bridge over the Fortran/C++ schism within the collaboration, because it should be possible for people on both sides of the divide to work with it easily. We define a language with a passing similarity to SQL in which we can compactly represent queries.

This introduction concludes with a simple example program in the higher-level language, which we will call μ QL. At this point, it should be read just to see the flavor of the relation-oriented approach. The example makes an autocorrelation plot for a single parameter number in the HITS bank:

```
// Give a name to this module
module autocorr;

// Define a table corresponding to a MIDAS bank created outside this module
import hits from HITS
  { double time, int parameter };

// Pick out only the S2 hits and make a table containing only their times
select from hits into s2
  where "parameter == 6000"
  { time };

// Make a table of all pairs of S2 hits that occur within 100 ns of each other
join s2 with s2 into s2_autocorr
```

```

coincidence time_1 with time_2 from 0 to 100;

// Plot these time differences in a histogram
histogram1d from s2_autocorr
  name "s2_autocorr" title "S2 Autocorrelation"
  "time_1 - time_2" bins 50 from 0 to 100;

```

The μ QL language in which this example is written is defined in detail in the rest of this document. It is processed by an automated tool into an equivalent C++ program. The translator performs a number of optimizations; not all tables mentioned in the input file are actually generated in concrete form.

2 μ QL Statements

2.1 `select`

The `select` statement makes a new table containing a subset of the rows and/or columns in the original table.¹

```

select from table1 into table2
  [ where "expression" ]
  [ column-list ] ;

```

- *table1* and *table2* are the original and output table names, respectively. They are not punctuated.
- where "*expression*": the expression selects which rows of the input table are to be copied into the output table. Any arithmetic expression that is valid in C should be acceptable here. The expression must be in double quotation marks.
- *column-list* means


```
{ [ type ] column-name [ = "expression" ] [ , ... ] }
```

It is a list of columns to include in the output table. For each column, at least the name is listed. Optionally, the name may appear with an expression that defines it in terms of the columns of the input table. If there is no expression, then it is copied directly from the column with the same name in the input table. Optionally, a data type may be included: it must be either `int` (integer) or `double` (double-precision floating point). If not specified, it defaults to `double` (for derived columns, defined by an expression) or to the type of the corresponding column in the input table (for columns that are directly copied). The column list is enclosed in curly braces.

¹Typewriter font is used to indicate a word to be typed literally, while *italics* should be replaced by an actual value. [Square brackets] indicate an optional element, but { curly braces } should be taken literally.

2.2 join

The `join` statement makes a new table by taking the θ -join of two input tables.

```
join table1 with table2 into table3
  [ coincidence column-name1 with column-name2 from min to max ]
  [ where “expression” ]
  [ column-list ] ;
```

- *table1* and *table2* are the names of the two input tables.
- The optional coincidence condition is nearly synonymous with the where clause where “*column-name1* - *column-name2* > *min* && *column-name1* - *column-name2* < *max*” However, it also asserts that *table1* is sorted in ascending order of *column-name1* and likewise that *table2* is sorted by *column-name2*. Consequently, it permits important optimizations to be performed. The column names are derived from the names of the columns in the input tables as described in the next item.
- The where clause selects rows in the output table just as it does in the `select` statement; together with the coincidence condition, it defines the θ in θ -join. In the expression, the columns of the two input tables are renamed as follows to avoid collisions:
 - If the column name is already unique within the two tables, it is not changed.
 - If joining two distinct tables, the column is renamed to *table-name_column-name*. That is, if the two tables are named `t0` and `hodoscope` and they each contain a column named `time`, the two columns are renamed `t0_time` and `hodoscope_time`.
 - If joining a table to itself, the two instances of *column-name* are renamed *column-name_1* and *column-name_2*.
- The *column-list* is much the same as for a `select` statement, except for the renaming rules described in the previous item.

2.3 import

The `import` statement associates a table with the contents of a MIDAS bank.

```
import table from “bank”
  column-list ;
```

The bank must have a four character name, following the MIDAS convention. Currently, it is necessary to supply a column list that describes the format of the bank. This column list is in the form used by the `select` and `join` statements, except that it is not permitted to include expressions in the list, only types and names. The author is thinking of ways to determine this list automatically, at least in requirement.

2.4 export

The `export` statement causes a table to be copied to a MIDAS bank where it can be used by other analysis modules, whether they are written in MQL or directly in C++:

```
export table to "bank" ;
```

2.5 Histogramming

There are six histogramming-related statements with very similar syntax:

```
histogram1d from table  
  [ where "expression" ]  
  name "name" [ title "title" ]  
  "expression" bins nbins from min to max [ title "axis-title" ] ;
```

```
histogram2d from table  
  [ where "expression" ]  
  name "name" [ title "title" ]  
  "x-expression" bins nbins-x from min-x to max-x [ title "x-axis-title" ]  
  "y-expression" bins nbins-y from min-y to max-y [ title "y-axis-title" ] ;
```

```
histogram3d from table1  
  [ where "expression" ]  
  name "name" [ title "title" ]  
  "x-expression" bins nbins-x from min-x to max-x [ title "x-axis-title" ]  
  "y-expression" bins nbins-y from min-y to max-y [ title "y-axis-title" ]  
  "z-expression" bins nbins-z from min-z to max-z [ title "z-axis-title" ] ;
```

The meaning of the fields in these expressions should be clear; they generate single one-, two-, and three-dimensional ROOT histograms, respectively. However, one often wishes to generate a series of related histograms at the same time. This capability is provided by variants of the statements with an additional "s" in the keyword and a `select` clause that specifies which histogram in the series is to be filled:

```
histograms1d from table  
  [ where "expression" ]  
  name "name" [ title "title" ]  
  select "selector-expression" from m to n  
  "expression" bins nbins from min to max [ title "axis-title" ] ;
```

```

histograms2d from table
  [ where “expression” ]
  name “name” [ title “title” ]
  select “selector-expression” from m to n
  “x-expression” bins nbins-x from min-x to max-x [ title “x-axis-title” ]
  “y-expression” bins nbins-y from min-y to max-y [ title “y-axis-title” ];

```

```

histograms3d from table1
  [ where “expression” ]
  name “name” [ title “title” ]
  select “selector-expression” from m to n
  “x-expression” bins nbins-x from min-x to max-x [ title “x-axis-title” ]
  “y-expression” bins nbins-y from min-y to max-y [ title “y-axis-title” ]
  “z-expression” bins nbins-z from min-z to max-z [ title “z-axis-title” ];

```

2.6 Ntuples

A ROOT ntuple is essentially an on-disk representation of a table. It may be generated using the statement

```

ntuple from table name “name”
  [ title “title” ]
  [ where “where-expression” ]
  [ “column-list” ] ;

```

The *column-list* is defined the same way as in the `select` statement. If it is omitted, then all columns of the table are copied to the ntuple.

3 Implementation notes

The μ QL-to-C++ translator is written in Java. The front end uses tools called JJTree and JavaCC that simplify the construction of parsers. JavaCC takes a formal description of the grammar for a language and generates a parser for that language. JJTree is a preprocessor for JavaCC that generates code to build a parse tree that corresponds to the input file.

After parsing the input, the translator makes several passes through the resulting tree. On the first pass, it locates and records the definitions of all tables. It then begins writing out the associated C++ code, beginning with a standard preamble and proceeding with various declarations and initializations. Finally, the code that implements the table operations described in the input file is generated, though not generally in the same order that it was specified in.

In most cases, it is possible to perform operations on the contents of tables as they are generated rather than first generating the table and then going through a second loop to use its data. Indeed,

the only tables that even need to be generated in concrete form are those which appear as *table2* in a `join` statement. These tables are generated with high priority as soon as the inputs upon which they depend exist, as in the following program:

$$\begin{aligned}A &= W \times X \\B &= Y \times Z \\C &= A \times B\end{aligned}$$

In this situation, the second statement $B = Y \times Z$ will be evaluated first, and the other two will be evaluated at the same time, with $C = A \times B$ nested inside $A = W \times X$.