CAP Current Implementation Guide

HPPC Department

July 6, 1995

Contents

This Current Implementation Guide is a companion to the CAP User's Guide. Where the latter defines the features that the CAP software will have, this guide states the features and behavior of the *current* implementation. It is intended as a reference for early users of the system.

We will attempt to keep this document as current as possible.

Recent changes will be placed in margin notes. Intentions for modifications to be done in the immediate future will be indicated in *italic* text.

Chapter 1

Data Mining on CAP

The objective of data mining on CAP is to start with a large collection of data on disk and tape, and scan through some portion of the data kept on disk at a high rate. Criteria are applied to this data, to select some small fraction of the events. Then the full (or a specified part of) data for the selected events are put into some desired format, and amalgamated into a result file. The intent is for the user to then mvoe those events onto a different system, for extensive analysis.

This is only one style of data analysis which can utilize the CAP system; we will ultimately support styles including n-tuple-based histogramming (PIAF or the like) and possibly a style based on incremental cuts, where information about the data selection is also part of the results, allowing for easy modification of selection criteria. However, the data mining style is currently the main thrust of the CAP project software efforts.

The data mining process proceeds in several steps. The user selection criteria are processed into a C++ program expressing the query; the program is compiled and run on a processor on the CAP system; the processor's data needs are served by I/O nodes connected to the body of disk data; the list of selected events is scanned to extract from tape any data needed to rebuild the full events; the selected events are built into a file in the desired format; and finally the user retrieves this desired file. The utilities guiding these steps—other than the creation of the C++ program implementing the query—are described in this chapter.

Prior to any data mining activities, the experiment has to have prepared in three major ways:

- A data model must be created, defining what the physics objects are for events in its data, and what attributes those physics objects have.
- The data in raw experiment form must be loaded onto the CAP system and "imported" into and object form based on that data model.
- The data in object form may be kept in tape storage, but the data representing all or some fraction of the events must be "migrated" onto the CAP system's distributed disks. Only data on disk can be scanned for event selection.

The CAP Experiment's Reference explains how each of these steps proceed.

Another prerequisite step is that the user must obtain an account on the CAP system for the purpose of submitting data mining queries. The CAP administration can be contacted to set up these accounts. Setting up the user account includes establishing several directories for the software to place data, code, and log files.

The user will typically submit query jobs from a remote workstation; there is no need to actually log in to the CAP system. The `.rhosts` file in the user's CAP area must allow access from his/her workstation; then quries can be submitted remotely. The product necessary on the remote workstation to do this query submission is called `capsqs`; it is distributed via the usual Fermiab ũps/upd mechanism, and runs on Unix workstations, including AIX and IRIX boxes. (It will also be possible to move query files to the user's area on the CAP system itself and submit queries from there. For instance, a user on a VMS workstation with no access to a Unix platform might choose this route.)

1.1 The SQS Utilities

Queries can be submitted to the CAP system from the user's local workstation, using the `capsqs` product installed on the user's machine. SQS stands for Submit Query Software; this product is available in the distribution directory `cap/distribution` on the CAP system. The user should do a

```
setup capsqs
```

before submitting queries. (Instructions for installing the product are presented elsewhere in this document.)

The steps in a user query are:

- Decide on a list of stores to scan for the query.
- Prepare the query file, describing the data to be scanned and event seletion criteria.
- Prepare a control "script file" guiding the query job.
- Submit the query.
- Monitor the progress of the query job.
- Retrieve the results: The selected events.
- These events, representing a small but (assumedly) interesting fraction of the total data, can then be used for more detailed analysis.

The following utilities are provided for performing these steps:

sqs_db [**-p** *prefix* [*token*]]
: Check the object stores to determine which events can be scanned.

sqs_submit *scriptfile* [**-b**]
: Submit a query prepared in MQL language into the CAP system for processing.

sqs_rm *jobid*
: Abort a query which has been submitted.

sqs_status [*jobid* — **-u** *user*]
: Show the status of a CAP query job or jobs.

sqs_view [*jobid*]
: An X-based tool to monitor the progress of a query job.

sqs_log *logfilename*
: A tool to access the job log file for the query job.

sqs_list [**-u** *user*]
: List all available query results stored a user.

sqs_get [**-u** *user* [-d *directory*] [-o *output_name*] [-m] *dataset_name*]
  Retrieve the query results from storage on the CAP system.

sqs_delete *dataset_name*
  Delete a query result from storage on the CAP system.

Before using these commands, the user must setup the product:

```
setup capsqs
```

1.2 The User's Area on the CAP System

As mentioned earlier, the CAP user must have an account on the CAP system. Although query jobs will typically be submitted remotely, and the user may well never actually log in to the CAP system itself, this account confers the privileges necessary to perform the various file accesses and process initiations inherent in a dta mining job.

Associated with the user account is a home directory area on the CAP system. Under this, there will be a subdirectory `sqsdir`. This subdirectory, in turn, has four subdirectories:

job
  The temporary home for the job sources, executables and Loadleveller's command files. After a job ends, the C++ files generated by MQL to execute the last submitted query will remain. These will have names derived from the query file name and the job id, with extension "`.cc`," e.g. `demo.query_1532.cc`.

log
  Contains detailed job logs for queries submitted. These will have names derived from the job id, e.g. `job_1532.log`.

output
  This area will hold output files produced by user functions; the `batch_userfile=<file>` line in a script file informs the query job that such a file is expected.

sqs
  Contains script and query files for queries submitted. These will have

names derived from the job id, e.g. `demo.query_1532`. This is a temporary home for these ifles; the next job submitted will clear out the existing files.

This directory structure is built automatically by the `sqs_inst` command. This environment is for sqs use only, although it is valid for users to read the contents of these areas. The user should not attempt to use those directories to submit queries directly, circumventing the usual mechanisms.

1.3 Searching the Stores Database: sqs_*db*

Allows a user to check all the object stores currently available for data mining on the CAP system. The output (stdout) of this tool is suitable for use in the data-specifier section of the query file.

sqs_db [-p *prefix*] [*token*] [-b]

-p prefix: Show stores only starting with the indicated prefix.

token: Show only stores including the token as part of their names. The current organization of data models has event-header stores einding in `EVE`. These event header stores are the onese which must be scanned; so this token is ususally specified in the `sqs_db` command.

Example:

```
sqs_db -p TOP_ EVE > myqueryfile
```

This will output all the names of event (`EVE`) stores that have the prefix `TOP_`.

A mandatory argument specifying the experiment/data model will shortly be added to the syntax of this utility. The data model name will be required to be supplied as the first argument.

1.4 Submitting Queries: sqs_submit

The primary means of submitting queries to the CAP system is the sqs_submit utility:

```
sqs_submit scriptfile [-b]
```

Queries written in the MQL language are submitted to the CAP system for processing. Two files are involved: a script file and a query file. The query file is described in the chapter on MQL.

The script file controls the job: Output format, batch behavior, notification of user, and so forth. Lines in the script file are space sensitive: *Whitespace on either side of the "=" sign is not permitted.*

The script file contains the following mandatory lines:

```
batch_model=<experiment-model>
batch_query_file=<file>
batch_output_format=<format>
```

- Specification of the experiment and data model to be used. This determines which universe of data is to be scanned. Within a given experiment, there may be multiple data models. The experiment-model specifies, in a single identifier, both the experiment and the particular model used, e.g. `d0model-2`.

  ```
  batch_model=D0model-2
  ```

- Specification of the file containing the query information (the data and critria parts of the query). There must be one such line in the script file.

  ```
  batch_query_file=myquery.query
  ```

- Specification of formats for the output file (or files) of selected events. *(Multiple formats are presently being implemented; it will be permissible to have more than one* `batch_output_format` *line.)*

  ```
  batch_output_format=<format>
  ```

  The formats accepted are model dependant. The *format* for the D0 models presently can be one of three options:

  object The extracted data remains organized as the C++ objects used to define the data model.

  zebra The extracted data is converted into the ZEBRA bank format—D0, for example, uses this as its native format, so the queries using the D0 data model support this option.

pointers The result is a list of persistant pointers to the selected events. This list can subsequently be used in further queries to define a sparse set of events to scan, or can be used to extract the selected events in a form to be determined later.

ntuple *The data model will shortly include an output format for creating n-tuples from selected events.*

A format might imply an extent of data also. (The extent determines which portions of a selected event are extracted.) *In d0model-2, it is intended that additional format options will be provided including—*

zebra-disk All the disk-resident data (including the cardinal data) of selected events is extracted; data is formatted as ZEBRA banks.

The script file also may contain one or more of the following optional lines:

```
batch_notify_user=<email-address>
batch_cardinal=<T|F>
batch_user_code=<file>
batch_user_include=<file>
batch_userfile=<file>
batch_class=<class>
```

- Specification of how to notify a user when each phase (selection, extraction, final results) of the data mining procedure occurs:

  `batch_notify_user = <email-address>` If this is omitted, no e-mail will be sent.

- Selection of the output extent to include only cardinal data. This will result in more rapid extraction of selected events, since the cardinal data for scanned events is always disk resident.

  `batch_cardinal=T` (Default is false, `F`—full data is extracted.)

- Specifying files containing sources that must be compiled and linked with the MQLgenerated code to produce the query executable. This is done when user functions are part of the query. The header file *must* contain proper function prototypes for any user function, so that the MQL can include that header and call the user function properly.

`batch_user_code=myfunct.f batch_user_include=myproto.h`

- Informing the system that the job will be producing user-controlled output files (for example, statistics files emerging from user functions). The system must be informed so that it can move the created files into the user's area on CAP after the job completes.

  `batch_userfile=myoutput` Multiple user output files may be specified on multiple lines.

- *Selection of batch class (priority, and so forth) to use for the query job. (Not yet implemented.)*

  `batch-class=A`

A sample script file:

```
batch_notify_user=joe@fnal
batch_queryfile=myquery
batch_output_format=zebra
batch_model=d0model-1
```

The `sqs_submit` command will provide a `jobid`, which can be used when monitoring or aborting the query, and which will identify the job result output files. If the system is unavailable for any known reason, the `sqs_submit` command will not process the query, and will provide the reason for the system being unavailable.

The `sqs_submit` command is by default intended to be used in interactive mode. If a query is to be submitted in the background or by a shell script, the `-b` option must be used.

Any error while submitting the job will be recorded at the logfile in the user's account in the cap system. The logfile's name and path will be given to the user in case of the query can not be submitted. The job receives a name when submitted to loadleveler (the job management software on the CAP system) and the user is immediately supplied with this name (which includes a number sometimes refered to as a jobid) to allow monitoring of the job status and access to error information.

1.4.1 Submitting Queries over the Web

There is a WWW page allowing a CAP user to submit a query in a fairly intuitive manner, to monitor its progress, and to retrieve results. This tool allows perusal of the data model's objects and their attributes, selection of available datasets, entering of the query itself, and obtaining access to error logs if necessary.

The *url* of this WWW page is `http://capcons.fnal.gov`.

Currently, the query entry step is purely an interactive activity; input from a prepared file is not supported. (Once the query has been submitted, the user can put it into the background, log off if desired, and retrieve results later on.) For queries involving particularly lengthy criteria, or to exercise SQS options not directly supported by the WWW page, it is recommended that the usual /tt sus_submit mechanism be used.

1.5 Monitoring Query Progress

There is a choice of tools for monitoring the progress of a query: One is text-based and can provide information about multiple queries; the other is X-based and provides dynamic information about a query's progress.

1.5.1 sqs_status

Information about the status of jobs submitted to process queries may be obtained by the `sqs_status` command:

> `sqs_status [-l] [`*jobid*`] sqs_status [-u` *user* `| -a] sqs_status -d` *numdays* `[-l` *jobid*`]`

With no arguments, `sqs_status` shows the status of all jobs submitted by the user which are running on the system. The options available are:

-l

The `-l` option specifies a display of the whole job log for a job—the history of each step taken. This can only be selected of a particular *jobid* is specified.

jobid

Selects a particular job of interest. The job id refers to the number output by the `sqs_submit` command.

-u The command can be made to display status of jobs submitted by a different user by the `-u` *user*.

-a This causes status for all jobs on the system, regardless of user, to be displayed.

-d Status information for the last *numdays* days may be displayed by the option `-d` *numdays*.

If the system is unavailable for any known reason, the `sqs_status` command will inform the user, and provide the reason for the system being unavailable.

1.5.2 sqs_view

This is an X-based tool to monitor the progress of a query job. The information displayed includes the lines sent for display by `sqs_status`, as well as dynamically updated information about percentage completed and fraction of events selected.

To run `sqs_view`, X11 software must be running on the workstation.

1.6 Retrieving Results

The query job involves a separate phase in which the data for selected events is collected and assembled into the desired result file. Upon sucessful completion, the results of a query (the extracted selected events) are kept on the CAP system. These results are a `tcache` dataset—the results are kept on the CAP system, on the hierarchical storage system, to avoid the problem of multiple users having very large collections of results cluttering the CAP disks. The `tcache` product implements space quotas, expiration dates, and so forth for data kept in an HSM tape system. (If the results are retrieved in a timely fashion, they will likely still be in the HSM disk cache, so the user will not have to wait for tape access.)

When the query is completed, the user is informed by a mail message, which supplies the name of the dataset containing these results; or the completion may be noticed if the job is being monitored.

The following tools allow the user to retrieve the results—access them from the CAP HSM and bring them to another system for further analysis.

1.6.1 sqs_list

This allows the user to see all available query results he has stored.

```
sqs_list [-u user]
```

1.6.2 sqs_get

The user retrieves the query results from storage on the CAP system by:

```
sqs_get [-u user] [-d directory] [-o output_name] [-m] dataset_name
```

-u
: May specify a different user.

-d
: Specifies the directory for the result file to be placed—the default is the working directory.

-o
: Specifies a name for the transfered result file—the default is the dataset name.

-m
: Specifies that after the transfer succeeds, the results file in `tcache` on the CAP system should be deleted.

dataset_name
: This required argument specifies which set is to be retrieved. A list of names can be found by the `sqs_list` command.

1.6.3 sqs_delete

Delete a query result from storage on the CAP HSM system. This can be used instead of the `-m` option to `sqs_get`.

```
sqs_delete dataset_name
```

Users are requested to avoid unnecessary clutter by deleting files no longer needed, or results which have already been transfered to remote workstations. However, quota and expiration date mechanisms will ensure that clutter does not become too severe.

1.7 When Things Go Wrong

A query job may not execute for any of a number of reasons, including:

- Missing scipt or query file.
- CAP system unavailable.
- Syntax errors in the script file, or in the query file.
- Use of objects or attributes in the query criteria which are not found in the data model.
- Syntax errors in the query criteria detected while compiling the query criteria.
- User function or header file needed for query criteria is missing.
- Required data sets not present.
- System problem, including loss of contact with a necessary I/O server node.

1.7.1 Availability of the CAP System

If the system is unavailable for any known reason, the `sqs_status` and `sqs_submit` commands will inform the user, and provide the reason for the system being unavailable. System administrators can declare the system unavailable (and provide the reason) for this information to be conveyed.

Unexpected problems, before they are detected, may sometimes cause a job submission to fail or a job to terminate abnormally. Such problems should be reported to the CAP administration.

1.7.2 Aborting queries

Once a query has been submitted, the user may discover that it will not execute properly. For example, upon monitoring the query, it may be discovered that 10% of all events are being selected, where the user expected .01%. Such jobs may be aborted and removed from the system:

`sqs_rm` *jobid*

1.7.3 Reading the Job Log: sqs_log

Detailed information about what occured in the course of a query job is kept in a job log file. This file logs all the actions taken by `sqs_submit` in behalf of the query job. This file is kept on the CAP system, under the user's home directory, in ˜*whoever*/sqsdir/log and will have a name like `job_1942.log`. However, you need not log in to the CAP system, or explicitly `ftp` the log file over to your workstation, to examine the log. Instead:

`sqs_log [`*logfilename*`]`

This will read the log file and send it to the user's terminal, piping through a "more" utility.

Output of the `sqs_log` command may be redirected into a file if needed. *We have to verify that this works properly, and is not fouled up by the "next page" mechanism.*

Also, we will support the more convenient syntax of `sqs_log jobid` *since the system can deduce the log file name from that.*

1.7.4 Job File Names

The results of the submitted job—the extracted events—will be kept on the Hierarchical Storage system as discussed in the section on retrieving results. And any output files produced by the user will end up in ˜*whoever*/sqsdir/output, with their designated names. Any other file related with the submitted job, created by the sqs utilities, will be identified by a job id assigned as the first step in the query submission (and reported back to the user by the `sqs_submit` utility). Examples of names look like:

loadleveler name received:	`capcons.549.0`
log file name:	`job_549.log`
Event pointers list file:	`list.job_549`

1.7.5 Examining the Query Code

Some user-induced errors might manifest themselves only when the code generated by MQL is compiled. The user can see the C++ code generated to implement a query, by examining the `.cc` file in ˜*whoever*/`sqsdir/sqs`.

The general structure of this file is that of a nested series of loops, based on walking through sets of persistent pointers to physics objects. The query criteria are found inside these loops.

Most errors, however, will involve attempts to use non-existant attributes or objects, or gross syntactical errors. These will be caught before the C++ file is to be compiled, and information about the error supplied to the user without needing to examine the query code.

1.8 Installing the SQS Utilities

There are two types of preparation needed to use the SQS utilities on a remote system. The tools must be brought over and installed on the system, and the user area must be prepared to have access to those tools.

1.8.1 Installing SQS on a System

1.8.2 Setting Up a User

Once the SQS utilities have been installed on a system, the following steps allow a user to submit CAP queries remotely, or to use any of the SQS utilities:

- The `.rhosts` file in the user's area on CAP must allow access from the remote workstation. For instance, if the workstation is fnmysgi, then `.rhosts` would include the lines

  ```
  fnmysgi.fnal.gov
  fnmysgi
  ```

- The user must do

  ```
  setup capsqs
  ```

  to set up the product. This line can be put into the user's `.login` or `.cshrc` file.

The SQS product uses the Unix commands `rcp` and `rsh`. If the user's shell is `csh(1)`, then these commands will not work properly if the `.cshrc` file unconditionally executes interactive or output-generating commands. The symptom of this problem will be a "`protocol screwup`" message. This can be avoided by putting the offending commands into the following conditional block:

```
if ($?prompt) then
<offending-commands>
endif
```

Then they won't won't interfere with `rcp`, `rsh`, or other non-interactive `rcmd(3)`-based programs.

Chapter 2

MQL - Writing Queries

The user executing a query on CAP starts by creating files describing the behavior of the query job, and the sets of data to be scanned and criteria under which an event is to be selected. These files are then submitted to the system using the `sqs_submit` utility described elsewhere in this document. The the idea of MQL data mining on CAP is that the user expresses selection criteria based on event data, and the data mining result is a file containing those events meeting the criteria, suitable for whatever dteailed analysis the user deems appropriate.

The present chapter discusses how to create these files controlling the job and expressing selection criteria.

(An alternative way of submitting queries involves a WWW utility. Nonetheless, it necessary to know how to express the query criteria, output formats, and so forth, as discussed here.)

2.1 What is MQL?

The CAP query processing is based on compiled (rather than interpreted) user input. MQL is the tool that translates user-supplied event selection criteria into a C++ program that will automatically be compiled and run on the CAP system. This program will search one or more specified sets of data contained in "stores" of objects on disk for events matching those criteria.

MQL written by Matt Braithwaite, originated as an improvement on the functionality of a prototype program called KQL, written by Kirill Denisenko.

KQL was based on PAW, although it is compiled rather than interpreted. MQL has significant extensions reflecting the fact that events have a more intricate data structure than flat n-tuples as per PAW or PIAF.

The data on CAP is organized as stores of various types of physics objects—electrons, jets, muons, and so forth. These objects have attributes (e.g. momoentum, energy, etc.); the objects and attributes are defined by the experiment in a "data model" (see the CAP EXPERIMENT'S MANUAL for details on how a data model is defined). The idea is that the user can "query" the data by supplying a set of criteria involving attributes of one or more physics objects. The MQL utility causes those criteria to be incorporated into a C++ program, which will be run (automatically) on the CAP system to execute the data scanning needed to select the events meeting the user's criteria. This generated program will implement all the tricky multiple looping implied by criteria which may involve multiple objects of one sort within an event. It will take advantage of the data organization by bringing in only the object data actually needed.

A later step extracts the whole (or a user-specified portion) of the selected events, and places them in a specified format for subsequent use (see the chapter on DATA MINING ON CAP) in further analysis.

2.2 The Script and Query Files

The MQL program runs on CAP, reading a query from standard input, terminated by end-of-file. This input is supplied by the `sqs_submit` utility; it comes from a file created by the user. This file tells the system which data to scan, what criteria to look for, how to present results, and so forth.

The `sqs_submit` utility uses a second "script" file to control the job: Decide on output format, batch behavior, notification of user, and so forth.

Here, we outline the parts of these files. The script file (usually named *whatever*`.script`) contains the control information. The query file (usually named *whatever*`.query`) contains two main sections, called the data and the criteria part.

2.2.1 Control Part: The Script File

The script file controls query execution, This file is processed not by MQL but by the `sqs_submit` utility. Therefore, the complete description of this file is given in the chapter on submitting queries, where the `sqs_submit` utility is described. Here, we will place a brief synopsis.

The script file contains the following mandatory lines:

```
batch_model=<experiment-model>
batch_query_file=<file>
batch_output_format=<format>
```

The script file also may contain one or more of the following optional lines:

```
batch_notify_user=<email-address>
batch_cardinal=<T | F>
batch_user_code=<file>
batch_user_include=<file>
batch_userfile=<file>
batch_class=<class>
```

2.2.2 Query File - Data Part

The first part of the query file specifies the runs to be scanned, that is, which sets of events the query is to scan, to select those events meeting the query criteria.

Every line in this part is recognized by a leading , and specifies a run or range of runs. Information about run names may be obtained via the `sqs_db` utility. The store name provided there and used in this file should be the event store, that is, the name including "EVE".

The syntax is:

```
@<runname | run-range>
```

to specify a `Run` or a range of `Runs`. If the name of a `Run` is of the form $< text >< number >< text >$ then a range of `Runs` may be specified in one line in the form $< text > [numberRange] < text >$ where the `numberRange` contains, separated by commas, numbers and/or ranges of numbers. For example,
`CHARM_[011452,011460-011485,011492]_EVE`.

2.2.3 Query File - Criteria Part

After the data part of the query file comes the criteria part. The data part is recognized by the fact that each line starts with "`@`".

In the criteria part, the following syntax rules are followed:

- Blank lines and whitespace may be inserted freely *except where noted*
- Comments in the C++ style are accepted: Everything on a line following `//` is treated as comment.
- Lines may be up to 1000 characters long. They may be extended to the next line using the Unix style continuation character: A line terminated by `\` extends to the next line. However, the 1000 character limit still applies to the total line, adding up continuations.

The criteria part specifies the selection criteria to be applied to each event in the specified data. The criteria are expressed as a series of conditions involving attributes of the objects defined by the data model, arithmetic and/or logical manipulations, and optionally user-supplied functions of the attributes.

A set of criteria consists of one or more lines each containing one criterion; the event is selected if and only if *every* criterion is satisfied.

A criterion is defined as a number of comparative expressions (which may be formed using the operators '`==`', '`!=`', '`<`', '`>`', '`<=`', and '`>=`') conjoined by logical operators (which are '`&&`' and '`||`'). A comparative expression or logical expression may be negated by enclosing it in parentheses and prefixing it with '`!`'. The operands of a comparison may be any arithmetic expression formed with numeric constants, the operators '`*`', '`/`', '`+`' (binary), and '`-`' (unary and binary minus), the result of a user function (see below), and the name of a data element (member) of an event (an attribute of an object belonging to the event—see below).

Note that the set of arithmetic operations does not include exponentiation (**). Non-trivial exponentiation may be expressed using the standard C function $pow(x, y) \equiv x^y$.

Warning: The negation operator means "negate this clause for *this particular set of objects tested.*" This will not have the same meaning as "true if this clause is *never* satisfied for any set of objects." For example,

```
!(electron.E > 20)
```

means the same thing as

```
(electron.E <= 20)
```

being statisfied for an event with at least one electron with E ¡= 20, rather than "accept this if there are *no* electrons with E ¿ 20 in the event." *If you mean the latter construct, that is awkward for now; the intent is to support the syntax* `electron#all.E <= 20` *for that meaning.*

As a special extension, comparing event attributes to single-quoted strings will also be supported (the event attribute is assumed to be a string, space-padded). This is not yet in place.

Each line of input must be a valid logical expression. However, multiple such lines are conjoined with '`&&`', that is, all lines of criteria in a set must be satisfied for the event to be selected.

Examples of the syntax provided below can be assumed to be selection criteria, that is, lines in the CRITERIA part of the query file.

Referring To Event Members

The physics objects that exist, and the attributes of each object, are defined by the experiment's data model. When a selection criterion involves an attribute of an object, that is refered to by $< object-name > . < attribute-name >$. For example:

```
electron.energy > 30

(muon.Pt > 20 || photon.energy > 20)

muon.Pt > muon.energy/2
```

An event may have multiple objects of the specified type; in that case, mql will create a loop over those objects, and the event can be selected based on any one of those objects. The user's criterion need not explicitly check that the event has at least one object of the specified type—MQL will properly skip events with no objects of the type in the criterion.

Note that MQL will behave in the intuitive manner for queries involving the OR of clauses referring to different objects. Naive implementations would check for the presence of at least one object of each type mentioned in a criterion line, and reject the event if there were no such objects in that event. However, the natural interpretation of clauses like

```
(electron.E > 10) || (muon.E > 12)
```

is to accept the event if it has an electon with energy ¿ 10 (or a muon with energy greater than 12) *whether or not it also contains a muon (or an electron).*

Elements of the event header, which would typically include the number of each type of object and some trigger information, may be refered to directly by name:

```
(trigger == 'high_pt') && (nelectrons < 4)
```

Referring To Event Members

The physics objects that exist, and the attributes of each object, are defined by the experiment's data model. When a selection criterion involves an attribute of an object, that is refered to by $< object-name > . < attribute-name >$. For example:

```
electron.energy > 30

(muon.Pt > 20 || photon.energy > 20)

muon.Pt > muon.energy/2
```

An event may have multiple objects of the specified type; in that case, MQL will create a loop over those objects, and the event can be selected based on any one of those objects. The user's criterion need not explicitly check that the event has at least one object of the specified type—MQL will properly skip events with no objects of the type in the criterion.

Warning:

MQL checks for the presence of at least one object of each type mentioned in a criterion line, and will reject the event if there are no such objects in that event. This has consequences in cases where the criterion line contains the OR of clauses involving two or more types of objects: If there are no objects of one type, then the event will not pass that criterion, *even though the object of another type may satisfy its clause.* For example:

```
(electron.E > 10) || (muon.E > 12)
```

An event with no muons *will not satisfy* this criterion *even if it has an electron with $E > 10$!* This behavior makes OR conditions involving attributes of multiple physics objects awkward. Improved behavior is defined in the MQL User's Guide and will be implemented.

Elements of the event header, which would typically include the number of each type of object and some trigger information, may be refered to directly by name:

```
(trigger == 'high_pt') && (nelectrons < 4)
```

Numbered Variables

Most data models permit multiple instances of a given type of physics object in a single event. For instance, an event might have several electrons, several muons, and so forth. MQL allows selection criteria involving multiple attributes of a single physics object, or of multiple physics objects of the same type in an event. The syntax is:

Criteria involving `object#n.attribute1` **and** `object#n.attribute2`
: This syntax specifies using the two attributes of the *same* physics object. Omitting $\#n$ is equivalent to using $\#1$; thus in the previous section, the example comparing muon Pt to energy compares the Pt and energy of the *same* muon.

Criteria involving `object#n.attribute1` **and** `object#m.attribute2`
: Assuming $m \neq n$, this syntax specifies using the two attributes of two *distinct* physics objects in the same event. If the event has fewer objects of this type than numbers specified (2 in this example), it cannot meet this criterion.

Criteria involving `object#0.attribute1` **and** `object#0.attribute2`
: Specifying $\#0$ rather than a positive integer relaxes the acceptance criteria to check against the attribute of *any* object of the given type. For example, one might want events with an electron of high Pt, and an electron (the same one or another one) of high rapidity.

Examples:

```
(electron#1.px > electron#1.E/2)
// Compares E and px of same electron

(electron#1.E > 20 && electron#2.E > 15)
// Requires two distinct electrons

(electron#1.px > electron#1.E/2) && (electron#2.E > 20)
// Requires a different electron with E > 20

(electron#1.px > electron#1.E/2) && (electron#0.E > 20)
// Requires any electron with E > 20 ---
// A single electron with px=15 and E=25 satisfies this
```

Compatibility Warning:
The #0 syntax and the convention that omitting the number is the same as #1 are confusing and will not be in the future (EQL) version of CAP. For forward compatibility, what we recommend for now is the following conservatiove approach:

- **If only one object of a particular type is used, feel free to use the form without a number (e.g.** `muon.E`**). This will have the same effect in** EQL **as at present.**

- **Avoid mixing the un-numbered form with numbered objects of the same type in the same query. These will have** *different effects* **under** EQL**.**

- **Avoid using the #0 "any" construct if possible, or realize that in a future version, thiese will no longer parse. (The future equivalent syntax will be that un-numjbered objects mean "any", which is the more sensible interpretation anyway.**

- **Avoid if at all possible queries involving the logical NOT operator (!). These are not going to be supported in** EQL **because the interpretations are sometimes ambiguous. We will attempt to bring the #ALL syntax into mql to provide the functionallity needed to do away with negation.**

Note that MQL takes care of setting up the loops and logic to test for the specified criteria. This flexibility is provided to cover all the typical cases. It is possible to contrive criteria which cannot be expressed elegantly in terms of this syntax; the user has the option of providing an explicit routine in lieu of the MQL query syntax.

Pointers to Other Objects

The data model can define attributes of objects which are *pointers* to other objects. The classic example is an object of type `jet`, which might contain pointers to one or more electrons, and/or one or more muons. The syntax for expressing criteria depending on such pointers is:

```
object1.object2.attribute
object1.object2.object3.attribute
...
```

For example, to find an event with a jet with an electron and a muon of high energy:

```
(jet.electron.E > 30 && jet.muon.E > 30)
```

The numbered variables convention may be used in conjunction with pointers, with the obvious meaning:

```
(jet#1.electron#1.E > 30) && (jet#1.electron#2.E >
20) \
&& (jet#2.electron.E > 20)
```

In criteria like the above, the user may be tempted to check that the jet has a sufficient number of pointers to meet the criterion. This check is not needed, as MQL inserts it automatically: In the above example, jets with less than two electrons would not be checked for the `jet#1` part of the critierion.

Aliasing

The query may also include, at any point in the criteria part, an alias definition of the form *name* = *expression*, where *expression* is an arithmetic expression as defined above. *expression* is substituted for subsequent occurences of *name*. This allows the user to avoid repeated use of messy expressions. It is useful, for example, when cutting on some expression lying in a specified range:

```
JetEleE = jet.electron#1.E + jet.electron#2.E
(JetEleE > 25) && (JetEleE < 30)
```

Standard math.h Functions

The criteria can, as above, contain references to numbers and to the variables defined by the experiment's data model as attributes of objects. They can also contain functions. The names of functions are passed through to the generated code verbatim, save that they are folded to lowercase. (This is so that users used to FORTRAN can write `SQRT()` in all caps and have it work). The arguments of these functions are treated just as if they were arithmetic expressions. `math.h` is included in the code produced by `MQL` so you can use the functions provided without having to do anything further. The $pow(x, y)$ syntax mentioned earlier makes use of a standard C function.

```
muon.E > (2*sqrt(pow(muon.px,2)+pow(muon.py,2)) + 4)
```

User Functions

Beyond the library of functions available through `math.h`, the user can create his own customized functions and use them as part of the query.

Variables used in the criteria are passed (by value) to the fucntion, in the obvious manner:

```
invarMass (muon#1.px, muon#1.py, muon#1.pz, muon#1.E, \
           muon#2.px, muon#2.py, muon#2.pz, muon#2.E ) > 5
```

Here the user function `invarMass` is passed eight arguments: px, py, pz and E of two (distinct) muon objects from the smae event. The user of a function does not invalidate or affect the automatic looping created by MQLİn the above example, the query code will make sure the event has two muons, and loop over pairs of muons in the event, calling `invarMass` for each possible pair.

The definition of the user function must be provided to the query process. This is done by a `batch_user_code=`$<$ *file* $>$ line and a `batch_user_include=`$<$ *header* $-$ *file* $>$ line in the script file. The contents of this header are C++ function prototypes for the user function; since users may be unfamiliar with C++ we will describe how to use this mechanism with user functions which were written in Fortran, C, and C++.

The data model for an experiment may define certain standard functions. While these are still "user functions," each individual user need not define them. Thus individual users need not become familiar with C++ to use the convenient syntax and the standard functions. In particular, it is likely that most experiments will define a function `invarMass` taking two object arguments, for each pair of object types for which taking an invariant mass makes sense. For functions defined by the data model, the individual user does not specify code or header files.

The Header File: Function Prototypes

The header file contains a function prototype, as described below:

In general:

A function prototype declares the name of the function, the type of value returned by the function, and the types of each of its arguments. For the purposes of query user functions, the function returns an `int` value. The arguments are whatever type the attributes might be; typically `floats`. The argument list may contain names for each argument, but this is not necessary.

Fortran:

Each argument should be either an INTEGER (*int* in C), a REAL (*float* in C), or a DOUBLE PRECISION (*double*). Also acceptable are INTEGER*2 (*short*) and BYTE (*char*), but these are less common in practice since data models tend to work with 4-byte numbers. The user function should return a value of one of those types.

The header file for a function called myfun, expecting three arguments—an integer, a real and a double-precision— and returning a real value, should look like:

```
extern "Fortran" float myfun ( int&, float&, double& ) ;
```

Since Fortran programmers may be unfamiliar with C++, some explanation and warnings are appropriate:

- The reason for this header file to be there at all is that the C++ compiler does strong type checking, that is, it verifies that the arguments passed to a function match the types that the function expects. To do this, every function must be declared and its return type and argument types specified in advance. The convention is to do this by including "header files" containing these declarations—"prototypes" of the functions.
- The `extern "Fortran"` tells the compiler to generate names suitable for linking in a Fortran function (rather than C++).
- The `float` preceeding the function name tells the compiler to expect a float (real number) return value. Warning: The function must return a value matching this; there is no protection against craziness happening if the user writes a Fortran function returning an integer, and the C++ code is told to expect a float.
- The `&` following each argument type tells the C++ compiler that arguments are passed "by reference" (that is, by putting addresses onto a stack) which is the way Fortran expects its arguments to be passed.

  > (Actually, MQL will generate code passing the specified attributes by value; but C++ is clever enough to see that the function prototype wants them by reference, and pass them in the appropriate manner.)

- Note the semicolon (;) at the end of the line declaring the prototype: This is mandatory. Warning: Omitting the semicolon will lead to a syntax error when the MQL generated code for the query is compiled, and since the statement is not terminated, the position reported for the syntax error may be after the actual erroneous line, in a non-obvious location.
- If the whole prototype does not fit conveniently on one line, it may continue on subsequent lines. Until the semicolon is reached, the statement is not considered terminated.
- More than one user function can be declared in a header file if necessary. Each has the format described above, starting from the "extern" and ending with a semicolon.

Non-numeric data types cannot be passed to a Fortran user function. Passing strings would be too awkward because of differences in the internal representation of strings between C and Fortran; passing structures or objects is too messy to contemplate. Using an argument as a one-dimensional array of numbers in the Fortran function is possible, as long as the user *knows* that the data model contains a one-dimensional array of the appropriate length at that attribute.

C:

Each argument, and the return value, can be of any valid C type. Arguments should generally be passed by value, although it is possible to pass a pointer to an attribute and know that subsequent attributes are in the expected place. (In particular, an array of attributes could be passed in this manner.)

The header file for a function called myfun, expecting three arguments— an int, a float and a double— and returning a float value, should look like:

```
extern "C" float myfun ( int, float, double ) ;
```

C++:

The most natural language for writing the user function is C++.

The function is a normal global C++ function, and the usual C++ prototype is used. (The function refered to in the query criteria must be global: Although you could imagine defining a class in the header file, and calling a member function of that class, this will not work in MQL because the "dot" operator will confuse the internal parsing into thinking this is a physics object, to be looped over for the event. Though the function named in the criteria must be global, it is then free to call members of various lasses, as per any C++ function.)

The code generated by MQL passes specified arguments by value. This includes entire objects, specified by naming *object#n* without an attribute—the whole object will be copied and passed by value. Of course, you can force passing an argument by reference merely by specifying that in the function prototype. An example:

In the header:

```
float myfun ( muon&, float& ) ;
```

And in the CRITERIA:

```
myfun (muon#1, electron.E) > 20
```

If necessary, a user function can use persistent pointers. For instance, a `special_energy` function of a jet object might dereference pointers to the electrons and muons associated with that jet. Or a function might invoke constructors which themselves derefernece persistent pointers. Individual users would tend not to create functions of that level of complexity. The experiment's data model creators might, however, and there is a caveat to be aware of: When creating a user function, avoid chains of ordinary persistent pointer dereferences which might cause early references to become invalid prematurely. The issues here are too involved to belong in this Users' Guide. Anybody creating such complicated user functions should first read the Fermilab Persistent Object Management Reference, particularly the section discussing physical pointers and lifetimes of persistent pointers.

Aside from implementing complicated calculations in a compact, organized and syntactically elegant manner, user functions can be exploited for their side effects. These can include creating and outputting histograms, for example. The way that extra output files would be written (as per the `extra_output` line in the control section is by a user function opening and writing to those files.

The user functions described here are required to have at least one argument (at least one atrribute or object passed in). In the near future, a "user subroutine" mechanism (detailed elsewhere) will be implemented. In that mechanism, for each event an array of objects or attributes is formed and provided to the user code—any looping over multiple objects of a particular type must be explicitly performed by the user subroutine. Queries containing user functions in MQL on the other hand, retain auotmated looping over multiple objects in an event; the presence of the function merely defines the way a particular criterion is calculated.

Variable Length Arrays

It is normal in any data model to have varying numbers of each type of object in a given event. Beyond that, it is possible to have variable length arrays in an object. The syntax for accessing the i-th element of an array X belonging to an object U would be:

```
(U.X(i) > whatever)
```

This requires, of course, preparation in the data model.

Note that this syntax uses parentheses (*á la* Fortran-style arrays) to specify the index, rather than square brackets. This stems from the way the data model sets up such arrays. In that model, X is not a usual attribute member of U. Instead, U has a **member function** X which extracts the appropriate item either from a (hidden) array of X-values at the end of the U object, or from a separate object in the same store consisting of a collection of X-values. In any case, X is a *function* associated with U, rather than a data element, thus the parnethesis systax.

Multi-dimensional variable-length arrays can be set up in a similar way; the data model defines the "extracting" funtions with two or more arguments. (Further details of this technique may be found in the CAP Data Model Creation Reference.)

Multiple Sets of Criteria

The current implementation does not support multiple sets of criteria producing independent output files.

The trick of defining independent sets of criteria with identical output files to work around the counter-intuitive behavior with respect to OR-ed clauses may or may not be supported at an early date.

Optimizing Your Query

MQL will evaluate the query criteria in a way which avoids obvious extra work. For example, if one clause of an AND is found to be false, then the other will not be evaluated. However, a certain amount guidance can make a vast difference in the performance of queries, particularly in determining how much data needs to be scanned to determine which events to select.

The rule is that (among a set of criteria) the criterion on the earlier line is evaluated before a criterion on a later line. These are to be AND-ed together, so if for a particular event line 1 fails, then line 2 is not evaluated. Similarly, within an AND on a single line, the leftmost expression is evaluated first. To take advantage of this, place the more stringent conditions first.

Since queries typically ultimately select a tiny fraction of events, it will commonly be the case that one or more clause will rarely be satisfied. The user will typically have a feel for which conditions are likely to fail in most events. If the condition most likely to fail is put first in the query, then the extra work having to do with evaluating the other conditions, and (probably more importantly) the extra I/O involved in bringing in the objects looked at in the other conditions will be avoided most of the time.

For example:

```
CRITERIA:
muon.usually_small > bigValue
```

```
electron.E > averageEnergy
horrendousUserFunction(muon.E) > typcialValue
```

will look through the muon stores, and only in the rare case of finding a muon with the `usually_small` attribute bigger than `bigValue` will it go on to examine the electrons and then maybe evaluate the horrendous user function. If the three lines were in the opposite order, it would evaluate the function, which often passes its condition; them pull in and examine the electron (and often pass that) and thus walk both the electron and the muon stores, doing a lot of extra work.

Naturally, expressions involving multiple numbered variables take longer to evaluate, because of the multiple nested looping necessary. (This is particularly true if several levels of looping are necessary, or if there are many objects of a particular type.) In that vein, there is an optimization which is typically obvious to users but which in general MQL will not catch automatically. This is illustrated by the following criterion:

```
electron#1.E > 10 && electron#2.E > 50
```

Here, in an outer loop, the electrons in the event are scanned looking for one with E ¿ 10. For each such electron found, the elctrons are again scanned (skipping the one found in the outer loop), looking for one with E ¿ 50. So if the first electron had E = 60 and the tenth E = 15, the first time through the loop the first electon would be used to satisfy E ¿ 10—the second condition will not pass. MQL will eventually get up to testing the tenth electron for E ¿ 10 and the first for E ¿ 50, and will thus eventually pass the criterion, but it will have done extra work. Instead, put the more stringent cpondition first:

```
electron#1.E > 50 && electron#2.E > 10
```

Another concern introduced by numbered variables arises from the fact that the loop over a numbered variable encloses all its uses. Thus, the following query:

```
ELECTRON#1.PX > 50 ... lots of stuff ... ELECTRON#1.PY > 50
```

is a really bad idea, because the "lots of stuff" will be executed for every choice of `ELECTRON#1`. Note also that the following is just as bad:

```
ELECTRON#1.PX > 50 ... lots of stuff ... ELECTRON#2.PY > 50
```

because loops over objects of the same type nest, so that the loop over `ELECTRON#1` will not close until after the loop over `ELECTRON#2`. Of course, loops over `#0` variables are exempt from all these contstraints. Simply stated, put uses of a numbered variable together, put numbered variables over the same object together, and put more stringent criteria first.

MQL does, as mentioned before, perform premature optimization, meaning that if the first half of an "and" clause is false, or if the the first half of an "or" clause is true, the second half will not be evaluated. This will even work with loops over numbered variables, in that premature evaluation of a logical expression will also provide an early jump out of the applicable loops.

2.3 Understanding the MQL Output

This section, which is not yet written, will include:

- Words about why one might want to look at the C++ MQL output.
- An explanation of how to get the MQL output to look at.
- An explanation of the various loop structures, qsets, and pptr concepts found in these files.
- Guidance in interpreting error messages.

2.4 WWW-guided Query Submission

There is a utility available on the CAP WWW server that allows people with CAP accounts to create and submit queries, guided by a graphical interface. The url of this utility is http://*whatever*.fnal.gov.

Using the WWW query submitter is straightforward and intuitive; most of the options described in the previous section can be specified in this form. It is nontheless recommended that serious users read the section on the query file, to acquire some understanding of MQL and learn what options are available and what syntaxes are supported.