SC2001/GriPhyN

Exploring Virtual Data Concepts in a Physics Data Reconstruction Application

Ian Foster `foster@cs.uchiago.edu`
Jens-S. Vöckler `voeckler@cs.uchicago.edu`
Mike Wilde `wilde@mcs.anl.gov`

10/02/2001

Abstract: This memo describes the demo database and access language using several different views.

1 The Database As Designed

Figure 1 shows the database desing using the object modeling technique. The actual implementation of the prototype will be described in section 2 (page 2).

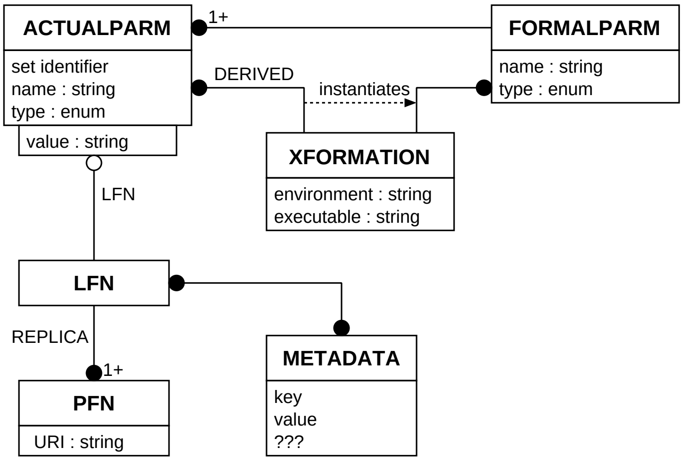

Figure 1: OMT of the database as designed.

Any instance of physical attributes, as captured in the `METADATA` class, can be used to describe one or more data file on a logical basis. Any logical filename `LFN` can be described by more than one metadata attribute. Unless we hear otherwise from the physicists, we need to assume a many to many relationship between these two classes.

Any logical filename `LFN` can be incarnated into one or more physical file `PFN`, as distinguished by its URI. The relationship between logical filenames and physical filenames is the replica catalog.

A logical filename is a special kind of `ACTUALPARAMETER` used when capturing the process context of a `TRANSFORMATION`, thus the optional multiplicity between `LFN` and `ACTUALPARAMETER`. Other parameters that are not logical files, include numerical or string values to a transformation. The value attribute captures either the logical filename, or the value of a non-file parameter. Thus the `LFN` does not contain any attributes, as it is just an specialization of a parameter.

Any `TRANSFORMATION` can be described by the executable and any number of related `FORMALPARAMETER`. A database implementation would need to model a transformation ID.

The interrelation between `TRANSFORMATION`, `ACTUALPARAMETER` and `FORMALPARAMETER` is the most complex, and expressed in three partially related relationships:

- A transformation is characterized by its parameters. The signature of the transformation includes input and output parameters, not limited to files. A transformation may have an arbitrary number of formal parameters, thus the 1:N relationship between `TRANSFORMATION` and `FORMALPARAMETER`.
- A transformation may have more than one derivation, using different values for the parameters, and as a set identified by their set-identifier, usually a number which is constant for the same related actual arguments.
- The relationships from a transformation to its formal and actual parameters are not independent of each other. Each instantiation of an actual parameter maps to exactly one formal parameter describing the entry, as shown by the dashed line. On the other hand, there may exist more than one instantiation of a formal parameter for the same function in different actual parameter sets, as shown by the N:1 relationship between the two parameter classes.

It may be the case that a ternary relationship between the three classes more aptly captures the interrelationships. The transformation end would have a 1 multiplicity, the formal argument a N multiplicity and the actual argument a M*N multiplicity.

From this perspective, it looks sensible to view the transformation and formal arguments as one block possibly called process to which one or more actual parameter implementations may exist. A view akin to this was taken in the prototypical implementation show below.

2 The Database As Prototyped

The first view deals with the database from a design and implementation perspective. Afterwards, the access language to the prototype is being described.

2.1 The Design View

Figure 2 (page 3) shows a OMT chart of the designed database as used in the prototype. Although it is unusual to model identifiers explicitly, for a database based implementation they make sense. Shown in color are parts which are not implemented.

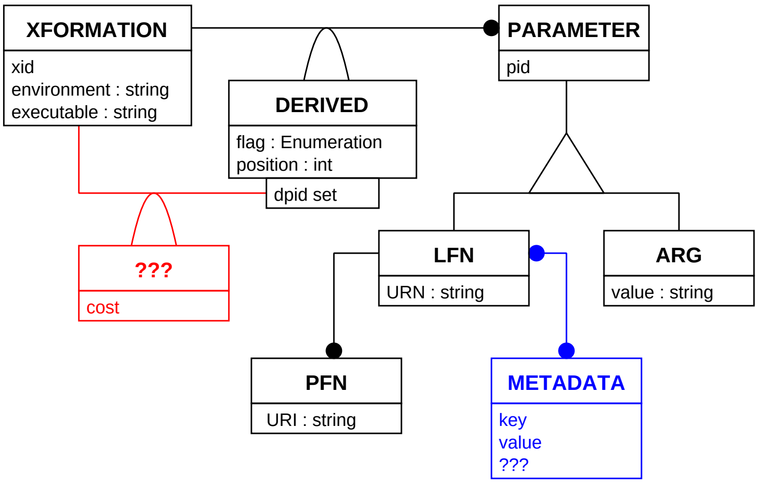

Figure 2: OMT diagram of table relations.

The demo database maintains the following tables.

`TRANFORMATION` describes transformation function for the execution of programs:

xid	ID
environment	string
executable	string

`PARAMETER` is the parent class to file-based parameters like logical filenames as well as non-file parameters like numerical and string arguments. As parent class, it provides only the parameter id common to its siblings:

pid	ID

`LFN` is the logical filename which describes a file argument:

urn	string

`ARG` on the other hand describes a non-file argument, e.g. a numerical or string argument to a transformation which is not a file:

value	string

`DERIVED` describes the relationship between abstract, formal `TRANFORMATION` and concrete, actual `PARAMETER`:

ddid	ID
flag	enum
position	integer

The pid and xid are implicit foreign keys. Together with the position they form the primary key. The derived data id (ddid), currently not implemented, allows to group sets of actual arguments.

The flag describes the kind of parameter used. It helps to distinguish between input filenames, output filenames, stdio filenames and regular non-file parameters. Each transformation will have at least one parameter for the resulting output of the operation.

TRANSFORMATION		
xid	**env.**	**executable**

(a)

DERIVED				
xid	**pid**	**ddid**	**flag**	**pos**

(b)

PARAM.	
pid	**value**

(c)

RC	
pid	**URI**

(d)

Table 1: The four tables in the current implementation.

A `PFN` helps to map an abstract logical filename to one or more connected physical filenames:

uri	string

The `METADATA` maps between physics attributes and respective logical filenames. Currently, metadata is not implemented, thus shown in blue.

Another piece of information may be gained by reusing the same transformation template for more than one set of actual parameters, thus the introduction of a ddid in the `DERIVED` table to denote a common set. For similar transformations, the machine type or some kind of cost could be modeled, shown as red relationship between `TRANFORMATION` and `DERIVED`. It is just a conceptual idea which might fit into our framework, but is not yet implemented.

2.2 Implementation-specific modifications to the design

In the database, the class hierarchy of parameters is modeled as just one table `PARAMETER`, because the types of the argument siblings overlap:

pid	ID
value	string

The relation between logical filenames and physical filenames is the replica catalog. The database schema simplifies the relation and class by defining a `REPLICA_CATALOG` table:

uri	string
pid	ID

The relationship between the classes `METADATA` and `LFN` is assumed to be many to many, thus the relationship needs to be explicitly modeled:

pid	ID
mda	ID

The physics related attribute sets need to be kept in a table of their own, for instance a `ATTRIBUTE` table, or something similar:

mda	ID
key	string
value	string
?more?	?type?

Thus, the current implementation is made up of the four tables shown in table 1.

3 Implementation of The Access Language

The virtual data access language contains a specification part which changes the database (like SQL modify, insert and delete) and a derivation part with reads the stored transformations (like SQL select).

From an abstract point of view, a functional specification of a transformation can in one stroke define all necessary components, e.g.

$$(o_1..o_N) = \text{function}[\text{spec}](i_1..i_M, p_1..p_O)$$

We found that this functional and atomic specification can be equivalently translated into a series of procedural steps which define the function and its related set of arguments, as shown abstractly:

```
start transaction
define function ...
define input file ...
define output file ...
define other arguments ...
commit transaction
```

The language described here is a ”quick hack” to get us going instead of implementing a well-designed context free grammar. An example will be provided in section 4 (page 10).

3.1 begin

```
begin xid
begin programname
```

With the `begin` statement, a transaction in the database is started, defining and changing each of the table as required along the way. If the program ends prematurely, the open transaction can be rolled back. Also, while the transaction is in progress, it will not be visible to other, simultaneously connected derivation clients.

Starting a transaction block requires to specify the transformation for which the block is used. The block itself defines the actual arguments to be used.

There are two forms to define a transformation. If the transformation id from the `TRANSFORMATION` table is known, it can be used as a shortcut for a previously specified transformation. It is not necessary to lookup unknown transformation - the arguments are matched to the database, and if the transformation template is known, it will be re-used.

When specifying a new transformation or when forcing (see below) a new transformation, some path to the executable, or its program URI, need to be specified. Everything beyond that is assumed to be part of the formal argument specification, which is stored in the template.

The formal arguments should contain a *$digit* placeholder for each logical filename and parametric argument (see below) that will be specified within the block.

```
begin /bin/cat
```

In the above example, `/bin/cat` will be assumed to be the program URI. Currently, environments are not supported, though this can be fittend into DAGMan.

3.2 Transaction Block Arguments

```
arg string
file i|o lfn
stdin lfn
stdout lfn
stderr lfn
```

Within a block, file arguments and non-file parameters for the transformation can be specified. The respective position in the actual argument list is implicitly given by the position in the transaction block.

Stdio argument do not have the notion of a position, but they have a singleton pattern to them: within any transaction block, each of `stdin`, `stdout` and `stderr` may be specified only once.

The non-file parameter value may contain whitespaces. The file argument uses a one letter direction flag for the file argument a logical filename, e.g. whether it is an input file or output file. Currently defined with a meaning are 'i' and 'o' for input and output files respectively. The translation of the logical filename to one or more physical filenames is done elsewhere.

```
arg -n
file i asdf
file i qwer
stdout zxcv
```

If a logical filename or parameter string already exists, the existing one will be used, unless a new incarnation is forced. If a dependency already exists, that is, the parameter id, transformation id, derived data id and parameter position acting as primary key already exist, no new dependency will be created unless forced.

The `file` parameter is actually a generic way to specify any type of argument. The possible set of characters and their meanings are shown in table 2. Unknown characters are not yet caught.

c	**type**
`-`	regular non-file argument
`i`	logical file, input
`o`	logical file, output
`I`	redirection of `stdin` from lfn
`O`	redirection of `stdout` to lfn
`E`	redirection of `stderr` to lfn

Table 2: Meaning of the type character.

3.3 cancel end

```
end
cancel
```

With the final `end` clause, the transaction can be committed. Alternatively, a transaction might be rolled back with `cancel`. Either `end` or `cancel` finish the transaction block opened by `begin`.

3.4 rc

```
rc lfn pfn
```

The `rc` fills the replica catalog. It allows for any logical file the specification of one or more physical files. The first existing physical file will usually be taken.

```
rc asdf /tmp/xx
rc qwer /tmp/yy
rc zxcv /tmp/zz
rc zxcv /tmp/ww
```

The mapping may be specified inside or outside a tranformation block. If specified outside, the new mapping will immediately be committed to the database. Within a block, the committment is delayed to the block finalizer. If the logical filename is known, its parameter id will be reused. If there are several to chose from, only the first will be used.

Forcing a logicial file name mapping creates a new instance (each time) to map to a physcial filename. I am not sure how useful that is. If the mapping already exists, it will be reused unless being forced.

3.5 Interlude Example 1

```
begin /bin/cat
  arg -n
  file i asdf
  file i qwer
  stdout zxcv
end

rc asdf file://tmp/xx
rc qwer file://tmp/yy
rc zxcv file://tmp/zz
rc zxcv file://tmp/ww
```

Assume that the above data is entered into the catalogs. Also assume that the file `/tmp/xx` and `/tmp/yy` exist, are textual, and readable. Calling the command `get zxcv`, it will eventually invoke `/bin/cat` with the appropriate arguments of `-n`, `/tmp/xx`, and `/tmp/yy` to produce a physical file `/tmp/zz`, which contains the contents of former files including the line numbers.

3.6 force

```
force begin ...
force arg ...
force file ...
force rc ...
force get ...
```

The `force` prefix before certain commands enforces the creation of duplicates. The specific nature is described in the section dealing with the command. Using the `force` prefix before a `get` command might *build* the required file instead of *makeing* it.

3.7 get

This command will be obsoleted in favor of the `dag` command. The `get` command is limited to local compute- and filesystems.

```
get lfn
```

Whereas the previously shown commands all deal with setting up the catalogs and manipulating the data, the `get` command retrieves the data, making use of the cataloged structures and dependencies.

```
resolve lfn into set of pfns
IF any pfn in set exists THEN return pfn
obtain pid for lfn
find first transformation for pid
   where it is an output flag
find all parameters for transformation
   ordered by parameter position in derived
FOREACH parameter
  IF parameter has input type THEN
     recurse with lfn
     add returned pfn to correct argument list
  ELSIF parameter has output type THEN
     resolve lfn into set of pfns
     add first? best? pfn to correct argument list
  ELSE
     add parameter to argument list
invoke actual transformation
return pfn from above
```

3.8 dag

The `dag` command allows to create a directed acyclic graph and matching job descriptions fitting for Condor from the interdependencies stored in the database. Currently, the complete DAG will be created. Future releases will mark those nodes for which data exist as done, and only generate complete rerun DAGs for forced execution.

```
dag basename lfn
```

The first argument provided is the base filename to be used to generate a DAG and to use for logging purposes. The DAG filename suffixes in `.dag` and the log filename in `.log`.

The second argument is the name of a logical file that needs to be recreated. From the dependencies stored, a DAG to create this file will be generated. Each node in the DAG will be numbered A, B, .., Z, BA, BB, .., ZZ, BAA, BAB, .. ZZZ, etc. For each node, a job submission file with suffix `.job` will be created.

```
obtain parameter id for logical filename
select all functions which contain this filename as output
obtain XID,DDID of function to use for creating the file
collect information about the transformation
collect all arguments for the actual transformation ddid
  order by position
FOREACH argument
```

```
    IF type is '-' THEN
      add value to parameter vector
    ELSIF type is 'i' or 'I' THEN
      recurse with input lfn
      add ddid to parents for argument
      obtain pfn for lfn
      add pfn to parameter vector or stdin
    ELSIF type is in ['o','O','E'] THEN
      obtain pfn for lfn
      add pfn to parameter vector or stdout/stderr
    ELSE
      complain
  generate node name
  mark node as visited
  generate DAG node submission file
  return ddid
```

Above Pseudocode shows the inner loop to create a DAG. In the current version, logical filenames are translated into physical filenames. It could be envisioned that with session arguments (see section 3.12), an abstract DAG using logical filenames can be constructed, since the current tool might not be the ideal place to make any mapping decisions.

```
open DAG file
IF file can be created THEN
  call dagger for given lfn
  dump job subscriptions to dag file
  dump dependencies from parents
ELSE
  complain
```

3.9 dump

```
dump rc
dump *
dump replica_catalog
dump parameter
dump transformation
dump derived
```

Dumps the content of the specified table in the database. Using an asterisk will dump the content of all tables.

3.10 debug

```
debug key level
```

Set the debug level for a certain key.

3.11 exit

```
exit
<EOF>
```

The explicit `exit` command or just an end of file condition exit the command interpreter. The user will be warned about outstanding transactions, which will be rolled back.

3.12 set unset show

```
set key value
unset key
show
show key
show key key ..
```

For future purposes, it might be feasible to declare session specific variables to modify program behavior. The `set` command allows to associate a key with an arbitrary value. The key must follow rules for identifiier specificiation in most computer languages.

The `unset` command releases a previously created association. With the help of the `show` command, you can inspect one or more specific, or all known associations.

3.13 save load

```
save pfn
load pfn
```

The `save` command at this writing only stores previously user-specified associations (see section 3.12) into a physical file. The associations are stored in the same way they would be typed in.

The `load` command allows to read those file-stored associations. Since the main even loop is employed for evaluation, a complete *include* functionality is provided by the `load` command. Nested includes are allowed, though circular dependencies are not checked (currently).

4 The Diamond Example

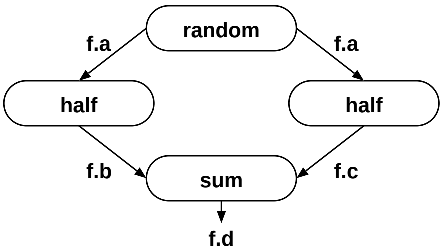

Figure 3: The diamond dag example from Condor.

The Condor diamond dag is an example for an acyclic directed graph of dependencies, see figure 3. A top node `random` generates an even random number. Two processes `half` calculate the half of the number. A final node `sum` adds the half. In the process, a number of logical (and, of course, physical) files is produced. The input files in terms of the command language are shown in figure 3 (page 11).

The first transaction defines the program `demo-random` which produces a result on `stdout` which in turn is capture in the logical file `f.a`.

```
#
# demo-02: the diamond dag
#
begin demo-random
  stdout f.a
end

begin demo-half
  stdin  f.a            # <=> file I f.a
  stdout f.b            # <=> file O f.b
end

begin demo-half
  stdin  f.a
  stdout f.c
end

begin demo-sum
  file i f.b
  file i f.c
  stdout f.d
end

rc f.a a.out
rc f.b b.out
rc f.c c.out
rc f.d d.out
```

Table 3: Definitions for the diamond dag in terms of the input language.

The second transaction reads `f.a` from `stdin` and prints its result on `stdout`. The third transaction uses the same program as the seconds transaction, thus employing the same transformation. Only the *derived data id* will change for the new set of actual parameters entered into the derived data table.

The final transaction takes two input files as regular files, which are passed on the commandline. The result is produced on `stdout`, and again captured in a file.

Finally, the translations from logical to physical filenames are defined. In future releases, it must be possible to describe the logical file in terms of a URN and physical files in terms of URIs. Though any logical file may map to more than one physical file, currently, only the first mapping is taken. In the future, a cost function must be employed to take the ”best” mapping. Figure 4 shows the content of a fresh database after inserting the diamond from table 3 (page 11).

Submitting the `get A f.d` command, five files will be created, containing the DAG submission and job submissions. The DAG and a sample node job submission file are show in table 5.

TRANSFORMATION		
xid	**env.**	**executable**
1		`demo-random`
2		`demo-half`
3		`demo-sum`

(a)

PARAM.	
pid	**value**
1	f.a
2	f.b
3	f.c
4	f.d

(b)

DERIVED				
xid	**pid**	**ddid**	**flag**	**pos**
1	1	1	O	-1
2	1	2	I	-1
2	2	2	O	-1
2	1	3	I	-1
2	3	3	O	-1
3	2	4	i	0
3	3	4	i	1
3	4	4	O	-1

(c)

RC	
pid	**URI**
1	a.out
2	b.out
3	c.out
4	d.out

(d)

Table 4: Content of the tables after inserting diamond.

```
Job B B.sub
Job C C.sub
Job D D.sub
Job E E.sub
PARENT B CHILD C
PARENT B CHILD D
PARENT C D CHILD E
```

```
# Filename: D.sub
#
Universe     = vanilla
Executable   = demo-half
Log          = A.log
Input        = a.out
Output       = c.out
Notification = NEVER
Queue
```

Table 5: submission files for the DAG (left) and node D (right).