Utility Scripts for the VSO Codebase

E. J. Mansky

1 Introduction

The VSO codebase at UCAR's HAO contains a number of utility scripts that provide additional functionality. Here we describe the utility script `mission_db.pl`, and the associated Moose packages and configuration files, that create and populate MySQL databases for use by the VSO.

The Perl script `mission_db.pl` is used to create the database schema for the meta-data needed on a per-DataProvider and per-instrument basis. The script also loads the meta-data into the tables. Once the meta-data is loaded into the database, the VSO codebase reads and uses the meta-data to construct and execute queries based upon the data in the meta-data tables.

The section "Usage" covers in detail the command-line invocation of the script, while the section "Requirements" details the CPAN modules required by the script and it's associated supporting code. The section "Moose Objects" provides details on the public and private methods in the Moose objects used by the driver script mission_db.pl. The section "Configuration Files" provides details on the layout and contents of the 2 configuration files driving the script mission_db.pl The section "Examples" provides details on case of the HAO instruments CHIP, K-Cor and CoMP.

2 Usage

The usage of mission_db.pl typically involves three steps, first is to create the database if needed. Secondly, create the tables in the database that will contain the meta-data. Finally, load the needed data into the tables.

A final, optional fourth step may also be executed, depending upon the DBA's preferences.

All the data needed to drive the script are in two configuration files: a schema file and a configuration file. Both are referred to herein as "configuration" files.

1) To create a database that will contain the meta-data used by the VSO codebase type:

```
mission_db.pl -mi vso -f vso_schema.tmpl -p $CONFIG_BASEPATH
              -st config -db_type mysql -m create -o database
              -u DB_USER -pw DB_PASSWORD
```

The creation of the database VSO will most likely not be necessary since it may already exist at your site. If so, the first step should be skipped.

2) To create the tables in the database created by the above command, type:

```
mission_db.pl -mi vso -f vso_schema.tmpl -p $CONFIG_BASEPATH
                    -st config -db_type mysql -m create -o table
                                        -u DB_USER -pw DB_PASSWORD
```

3) To load the database tables with the required meta-data type:

```
mission_db.pl -mi vso -f vso_schema.tmpl -p $CONFIG_BASEPATH
                    -st config -db_type mysql -m load -o table
                                        -u DB_USER -pw DB_PASSWORD
```

where DB_USER and DB_PASSWORD represent the DB User and Password in the MySQL database, respectively.

The environment variable `$CONFIG_BASEPATH` points to the directory where the two configuration files are located. A second environment variable, `$LOGPATH` points to the location where log files will be written during processing.

The final step of loading the data into the VSO tables used to store the meta-data needs to be executed only when a stored procedure changes, which would be a result of a DB schema change, or, with the addition of new functionality. Generally the frequency of such schema changes is low, so the loading step in most cases would be done only once.

4) To load the SQL commands associated with the named stored procedures above, into their respective databases, the optional fourth step may be executed:

```
mission_db.pl -mi acos -f acos_schema.tmpl -p $CONFIG_BASEPATH
            -st config -db_type mysql -m load -o table  -sp_only 1
                                        -u DB_USER -pw DB_PASSWORD
```

for the example of the ACOS database at HAO. See §5 for details.

2.1 Full Usage Statement

The complete Usage statement of the script is:

```
mission_db.pl -mission | -mi (Mission) -file | -f (Filename)
   -path | -p (Path) -schema_type | -st (Type of schema [ flat | config ])
                                -db_type (Type of DB [ mysql | postgres ] )
                        -mode | -m (Mode of operation [ create | drop | load ] )
                                -option | -o (Option [ database | table ] )
                -sp_only | -sp (load only stored procedures in non-VSO DB)
                                                -user | -u (DB User)
                                            -password | -pw (DB Password)
        -host | -h (DB Host)  -optional, may be placed in configuration file
-privs | -pv (DB Privileges)  -optional, may be placed in configuration file
                    -help | -h 1 : This message, 0: default is no help message
```

3 Requirements

The following CPAN modules are required to install and execute `mission_db.pl`:

```
Moose
Config::FromHash
Tie::IxHash
Clone
DateTime
```

The key package is Moose since the objects used by mission_db.pl are all Moose objects. Newer installations of Perl after 5.18 should already have Moose installed.

Another key CPAN module is Config::FromHash which most likely will need to be installed. Config::FromHash is used so that both the configuration and schema files can be written as arbitrary Perl hash structures and read into the code directly as such.

4 Moose Objects

The Moose framework is used for the core Object `MissionDB` since it provides several useful features over the older OO Perl framework. No prior experience with Moose is necessary to run the script, only that Moose itself is available in the Perl being used.

The first advantage Moose provides is automatic constructors and destructors for a given Object. Secondly, Moose provides automatic type checking of all attributes associated with a given Object. Attributes are special keys associated with the Object itself; and, if, at the time of either Object creation or use, depending on the setting of the Boolean variable `lazy`, an attribute does not have the correct type of value, an error is thrown. Finally, Moose has a large number of extensions, two of which: `before` and `after`, are especially helpful. `before` and `after` names methods that fire *before* or *after* the indicated method, respectively, thereby providing an easy technique to build dependencies among a set of methods and attributes.

4.1 Purpose

The purpose of `MissionDB` is to provide a single Object that will create, drop or load tables for a given Mission's DB based on a schema file. The Object provides one public method, "execute" to do all of the given actions available. Which particular action is executed depends on the settings of the required options at the time the Object is created.

A second public method, "validate", performs a validation check of the tables that contain file-related data to see if all the expected data was loaded or not.

The driver script, `mission_db.pl`, creates the `MissionDB` object first from the data passed in on the command-line. Then the public methods `execute` or

validate are called from the MissionDB Object by the driver script to do the desired actions of create or load.

4.2 Initial Usage *–for Object set-up*

```
my $missionObj = MissionDB → new({schema_type => $stype,
                                  schema_path => $spath,
                                  schema_file => $sfile,
                                      db_type => $dbtype,
                                      mission => $mission,
                                         mode => $mode,
                                       option => $option,
                                   '_db_user' => $dbuser,
                                     '_db_pw' => $dbpw,
                                   '_db_host' => $dbhost,
                                  '_db_privs' => $dbprivs,
                                  });
```

where the values for each key are defined locally in the caller, appropriately.

4.3 Subsequent Usage *–to execute the desired action*

$missionObj → execute *–for create or drop*
$missionObj → validate($data) *–for validation*

where $data = data to be loaded/validated. See below for details.

4.4 Attributes

schema_file

The attribute schema_file contains the filename of the schema file defining the DB to be created and loaded.

schema_path

The attribute schema_path contains the path to the schema file defining the DB. Currently the parent caller defines the path via a shell environment variable CONFIG_BASEPATH. Other mechanisms will of course work equally well.

schema_type

The attribute schema_type is a string with a value of "config" if the schema file is general hash in a flat file. In the future, as the need arises, other types of schema files will be added.

schema

The attribute schema contains the parsed results of the schema file as a set of nested, tied hashes. This attribute is the main output of the private function _get_schema.

did_schema

The Boolean attribute did_schema is set to 1 the first time the schema file is read and parsed.

db_type
The attribute db_type is a string, with a value indicating the type of DB to create. Currently only MySQL is supported. Future support for PostgreSQL is planned.

mission
The attribute mission is a string with a value indicating the specific mission for which a DB is being created and loaded for.

mode
The attribute mode is a string with a value of "create", "drop" or "load" that indicates which action to take in the database.

option
The attribute option is a string with a value of "database" or "table" to indicate which specific DB object to perform the action specified in mode upon

sp_option
The optional Boolean attribute indicates that only the stored procedures in the schema file are to be loaded into the specified database.

_db_user
The attribute _db_user is the User to login as into the DB.

_db_pw
The attribute _db_pw is the Password for the User one is logging into the DB as.

_db_host
The attribute _db_host is the Host where the DB is located. Currently the DB resides on the same machine as the Apache instance the data is being loaded for, so the usual localhost IP address is used for the value. That will need changing if the layout changes.

_db_privs
The attribute _db_privs is a string with a value indicating the DB privileges to use for each action. Currently set to "admin".

4.5 Methods

4.5.1 Public Methods

execute
The method `execute` performs the desired action, given in mode, on the target DB object in option, using the the DB schema specified in the three variables schema_file, schema_path and scheme_type.

The overall flow in `execute` is to first get the schema by calling the private method `_get_schema`. Then a connection is made to the specified DB and the

desired SQL command is constructed and executed. During the loop over the tables a determination is made as to which tables have already been operated on and are skipped, allowing only the tables that need the operation to be worked on. Finally, the results of the actions taken are printed to the log file.

validate

The method `validate` performs a check of the actual no. of rows loaded into the DB for the tables with file-related data and compares against the expected no. of rows. Success or failure log messages are generated in either case, and printed to the log file.

4.5.2 Private Methods

`_get_schema`

The method `_get_schema` reads the schema file, using the CPAN routine Config::FromHash, and parses the results into tied hashes, using the CPAN routine Tie::IxHash, so that the order of the table and column keys are preserved. Once the tied version of the schema hash is completed, it is saved for later use by the object in the attribute schema and the corresponding Boolean attribute did_schema is updated, so that the schema is read and parsed only once upon the call to execute, NOT at object initialization.

`_getDBConnection`

The method `_getDBConnection` gets the 4 DB related login variables, _db_user, _db_pw, _db_host and _db_privs from the object and establishes a connection to the DB by a call to _DBlogin. The connection is returned.

`_DBlogin`

The method `_DBlogin` logs into the type of database requested, with the specified User and Password. The connection is returned.

`_determineTableState`

The method `_determineTableState` executes a SHOW TABLES SQL command in the specified database and returns the list of tables returned. The returned array, containing table names, is used to determine if a given table has already been created, and if so, skip it.

`_determineTableLoadState`

The method `_determineTableLoadState` executes a SHOW TABLES SQL command to get a current list of tables in the DB to test against. Generally, there will be two types of tables in a given Mission DB, one set of tables dealing with the instruments and detectors and independent of the data files of the mission. The second set of tables store information on a per-file basis. Such data includes filenames, file mod dates, file sizes and checksums. The two types of tables are distinquished by the optional Boolean key per_file in a given table's hash in the schema file. For the tables with per_file = 0 (instrument-related tables), an immediate check of the row count is done and compared against the expected. For the tables with per_file = 1 (file-related tables), a check is deferred

until deferred_run = 1 is passed in the Hashref $data. Once deferred_run = 1 is seen, then the file-related tables actual vs. expected row count check is done.

After the checks of the tables are done a Hashref of the Boolean status of each check, on a per-table basis, is returned.

`_mapKeys`

The method `_mapKeys` maps the columns declared as keys in a given table in the schema file into a hash, $ckeys that is ordered by the key name, table name and sort_order of the key. The reason for the particular ordering is to make the subsequent generation of the SQL commands to create a table and multi-part keys as easy as possible.

The hash $ckeys, and the last column in the given table, $last, are returned.

`_mapType`

The method `_mapType` is called from inside a map statement where it is passed, the current table name, that table's column schema and the output of `_mapKeys` –the last column name of the current table and the hash $ckeys of the keys for that table. The return value is a sequence of strings of each column type and lastly the Primary Key and Secondary keys (if any). The parent caller then uses a join of the results of the map statement to construct the string that will form the CREATE TABLE command.

`_getColMap`

The method `_getColMap` constructs a sequence of INSERT SQL commands to load the data, passed in $data from method execute, into the DB for a given table. Two key helper functions, `_getColValues` and `_getColVariables` are used to construct the two pieces that go into the INSERT command. The row count of the data loaded is returned.

`_getColValues`

The method `_getColValues` constructs and returns an array of placeholders, one for each column in the given table. The parent caller then joins the array elements together into a string for the VALUES part of the INSERT command.

`_getColVariables`

The method `_getColVariables` constructs and returns an array of values to be loaded into the DB for a given table, one element per column. The returned array is then stepped through and a `bind_parm` is called for each value. This is the key method in the loading process and calls the `MissionLoader::$Mission` plugin `getLoaderVariables` to obtain the mission-specific data. Note that the particular MissionLoader package used is not known until runtime, so a require is used instead of use. See the perldoc for MissionLoader::EUNIS for additional details.

`_logResults`

The method `_logResults` generates printf statements, using the attribute log_file for the filehandle to determine which file to print to. The overall purpose of `_logResults` is to encapsulate as much as possible the print statements to a single method.

4.6 Configuration Files

Two files are required to define all the variables needed to create and load the meta-data VSO database tables. The database schema for the VSO is defined in the hash in the file vso_schema.tmpl. The meta-data to be loaded into the VSO tables is defined in the corresponding configuration file, vso_config.tmpl

4.6.1 Schema File

The schema file is a hash with required keys and optional sub-hashes that define all the data necessary to create, drop or load a DB for given mission.

The tables, columns and stored procedures are all created in the order they appear in the schema file by the use of the CPAN module Tie::IxHash and the use of the arrays `ordered_tables`, `ordered_columns` and `ordered_procedures`, respectively.

Required Keys

database

–value is the name of the database to be created, dropped or loaded into.

ordered_tables

–value is an array containing the same table names as in each table hash. The tables will be created, dropped or loaded in order they appear in the array.

tables

–subhash containing keys, one for each table to be acted upon. In turn, each table key points to a subhash that then should contain a subhash of column names. Each column subhash then should contain keys: type, key and optionally key_names. The key subhash has one of three possible values: primary, secondary or both. Primary indicates the column will be the primary key of the corresponding table. Similarily for secondary. If the column is going to both be a primary key for the table and be part of a secondary key, with potentially other columns, then it will also be a secondary key, which is indicated by the value both.

For secondary keys an optional subhash key_names can be used to define the name of the secondary key and the order the column should appear in the secondary key. In the example below col2 of table_2 will appear as the fifth key in the secondary key sec_key_name.

Also required in each table subhash is a key: ordered_columns. The key ordered_columns is an array containing the column names in the order to be created, dropped or loaded in the DB. The array elements should of course match the keys in the columns subhash.

```
tables => {
 table_1 => {
  ordered_columns => [col1, col2, ..., colN],
  columns => {
   col1 => {
```

```
      type =>' DBtype',
     },
     col2 => {
      type =>' DBtype',
      key =>' primary',
     },
...
     colN => {
      type =>' DBtype',
     },
    },
   },
   table_2 => {
    ordered_columns => [col1, col2, ..., colN],
    columns => {
     col1 => {
      type =>' DBtype',
      key =>' primary',
     },
     col2 => {
      type =>' DBtype',
      key =>' secondary',
      key_names => {
       sec_key_name => {
        sort_order => 5,
       },
      },
     },
...
     colN => {
      type =>' DBtype',
     },
    },
   },
...
   table_N => {
    ordered_columns => [col1, col2, ..., colN],
    columns => {
     col1 => {
      type =>' DBtype',
     },
     col2 => {
      type =>' DBtype',
      key =>' primary',
     },
...
```

```
    colN => {
     type =>' DBtype',
    },
   },
  },
}
```

Optional Keys

If stored procedures are needed in the DB, then two additional keys are needed.

```
ordered_procedures => [stored_proc1, stored_proc2, ..., stored_procN],
stored_procedures => {
 stored_proc1 => {
  script => qq{
     SQL source code for stored procedure #1
  }
 stored_proc2 => {
  script => qq{
     SQL source code for stored procedure #2
  }
...
 stored_procN => {
  script => qq{
     SQL source code for stored procedure #N
  }
 }
}
```

4.6.2 Configuration File

The configuration file vso_config.tmpl contains a hash, VSO, with the data to be loaded into the 5 tables in the VSO database. The VSO hash is composed of multiple keys that are used in MissionDB.pm to load data into the DB and later validate the loaded data.

Here is a summary of the most important keys and their use in the VSO hash:

per_year Boolean variable. Set to 1 for Missions that have data on a per-year basis. Set to 0 for VSO.

per_site Boolean variable. Set to 1 for VSO. Used during validation checks of data loading.

stored_procedures Array variable, the elements of which are the names of the stored procedures that are in the vso_schema.tmpl in the sub-hash stored_procedures.

DataProviders Key sub-hash containing the data to be loaded into the table `data_providers`.

Credentials Key sub-hash containing the data to be loaded into the table `data_feed_credentials`.

Mapping Key sub-hash containing the data to be loaded into the table `schema_mapping`.

SwitchoverDates Key sub-hash containing the data to be loaded into the table `instrument_details`.

5 Examples

To illustrate the use of `mission_db.pl` in a specifc case, we highlight here the usage of the meta-data needed for the HAO instruments CHIP, K-Cor and CoMP. The HAO instruments provide a good use case for the problem of when one group of instruments is split between two databases, each with a distinct schema, and a second group of instruments is located in only one DB.

To properly encapsulate the problem of a given instrument's data being distributed among multiple DBs, while other instrument's data is *not* distributed, we added a new variable, *instrument group no.*, represented in Perl as `$igroup`, with an associated column in the VSO tables, `IGROUP`, both of which are integers. By using an integer variable to do the mapping for multiple DBs, we avoid enumerating lists of specific instruments in the main Perl backend code `MLSO.pm`. Using an integer for the mapping also provides an easy way to add new instruments in the future.

The number of instrument groups associated with the HAO is set by the config key `no_igroups` in the hash `DataProviders`:

```
VSO => {
 per_year => 0,
 per_site => 1,
 completion_criteria =>' DataProviders',
 login_criteria =>' Credentials',
 instrument_criteria =>' SwitchoverDates',
 mapping_criteria =>' Mapping',
 key_for_id_per_key =>' Sources',
 stored_procedures => [read_all_dps, read_dp_credentials, read_swover_dates,
                       read_mapping', read_sp_details],
 DataProviders => {
  HAO => {
   Sources => {
    MLSO => {
     data_feed_type =>' remote',
     detail_tree_type =>' INSTRUMENT',
     no_igroups => 2,
     metakey =>' YYYY',
     plugin_name =>' hao_instrument',
     data_page_available => 1,
     active => 1,
```

```
      },
     },
    },
   },
  },
```

Snippet from `vso_config.tmpl` illustrating the key `no_igroups` in input data hash `DataProviders`. The remaining variables are used in error message generation and data validation, both of which are described in other documentation.

The instrument group no. is input as nested keys in the data hash `Credentials` as follows:

```
Credentials => {
 HAO => {                    # Provider
  MLSO => {                  # Source
   'ACOS' => {               # Database name
    '1' => {                 # instrument group no.
     LOGIN =>' vso',
     PW =>'!letmein!',
     HOST =>' mlso.hao.ucar.edu',
     PORT =>' 3306',
     MECHANISM =>' DBI',
     DB_TYPE =>' mysql',
     SP_NAME =>' query_acos',
     ARG_LIST => [nqyear, time_start, time_end, instrument, nqpostfix],
     SQL_LIST => [SQL_INTEGER, SQL_DATETIME, SQL_DATETIME, SQL_VARCHAR, SQL_VARCHAR],
    },
   },
   'MLSO' => {
    '1' => {                 # instrument group no.
     LOGIN =>' vso',
     PW =>'!letmein!',
     HOST =>' mlso.hao.ucar.edu',
     PORT =>' 3306',
     MECHANISM =>' DBI',
     DB_TYPE =>' mysql',
     SP_NAME =>' query_mlso',
     ARG_LIST => [time_start, time_end, instrume, telescop, nqpostfix],
     SQL_LIST => [SQL_DATETIME, SQL_DATETIME, SQL_VARCHAR, SQL_VARCHAR, SQL_VARCHAR],
    },
    '2' => {
     LOGIN =>' vso',
     PW =>' different_one',
     HOST =>' databases.hao.ucar.edu',
     PORT =>' 3306',
     MECHANISM =>' DBI',
     DB_TYPE =>' mysql',
```

```
        SP_NAME =>' query_mlso',
        ARG_LIST => [time_start, time_end, instrume, telescop, nqpostfix],
        SQL_LIST => [SQL_DATETIME, SQL_DATETIME, SQL_VARCHAR, SQL_VARCHAR, SQL_VARCHAR],
       },
      },
     },
    },
  }
```

where the data in the `Credentials` hash is loaded into the VSO table `data_feed_credentials` which has a 4-part index: `PROVIDER`, `SOURCE`, `IGROUP`, `DB_NAME`.

The VSO codebase then uses the stored procedure `read_dp_credentails`:

```
  read_dp_credentials => {
   script => qq{  CREATE PROCEDURE read_dp_credentials()
                  DETERMINISTIC READS SQL DATA
                   BEGIN
                      DECLARE RV INT;
                      SET RV = 0;
                      SELECT * from VSO.data_feed_credentials;
                   END;
                },
  },
```

to retrive the data in the table during startup and load it into a 4-dimensional hash `$db_credentials`:

```
  $dp_credentials → {$dp_row{provider}}{$dp_row{source}}{$dp_row{igroup}}{$dp_row{db_name}} = {
             'LOGIN'     => $dp_row{login},
             'PW'        => $dp_row{pw},
             'HOST'      => $dp_row{host},
             'PORT'      => $dp_row{port},
             'MECHANISM' => $dp_row{mechanism},
             'DB_NAME'   => $dp_row{db_name},
             'DB_TYPE'   => $dp_row{db_type},
             'SP_CNAME'   => $dp_row{sp_cname},
             'SP_QNAME'   => $dp_row{sp_qname},
  }if $sp = read_dp_credentials && defined($dp_row{igroup});
```

where the 4-dimensional key is the Perl counterpart to the 4-part DB index for the corresponding table. The two stored procedures in `SP_CNAME` and `SP_QNAME` are loaded into the database indicated by the key `DB_NAME`. To provide a complete self-contained set of codes and config files, the loading of the stored procedures in the ACOS and MLSO DBs can be accomplished by executing step #4 in §2 using the provided schema files `acos_schema.tmpl` and `mlso_schema.tmpl`.

Similarly, each of the remaining VSO meta-data tables has a stored procedure that is used to read data in from that table, and a corresponding Perl multi-dimensional hash that stores the data in the most appropriate manner for use in `MLSO.pm`. The result is a general mechanism that supports the use

of stored procedures on a per-DataProvider/per-instrument basis in the VSO codebase so that on-the-fly hashes can be constructed that provide the names of the DBs and stored procedures to query and call. during a given VSO search. The resultant Perl code in `MLSO.pm` is thereby simplified so that no details of specific stored procedures argument lists need be in the code explicitly, for example. Such details now reside in the meta-data tables and the stored procedures themselves.

Tab. 1: Mapping of Instrument to Instrument Group number

Instrument	Instrument Group Number	Comment
CHIP	1	Data in ACOS & MLSO
		PER_YEAR = 1 in ACOS only
K-Cor	2	Data only in MLSO

To support the new code that uses the stored procedures in ACOS and MLSO databases to count and query the HAO data, 5 new methods have been added to the backend package `MLSO.pm`: `getSPNames`, `getNumFound`, `getSPArgs`, `mapField` and `getRecords`.

5.0.1 `getSPNames`

The public method `getSPNames` first retrieves the VSO meta-data from the Config Object, and then constructs the array-of-hashes structure, `$sp_names`, that contains the DB, the stored procedure names that execute the counting and querying for a given instrument, together with the argument and SQL type lists for each stored procedure.

To simplify the logic of which databases are to be used for a given query, the start and end dates of the User's query, together with the switch-over date of the given instrument (if defined) are all first converted to DateTime objects. Then, depending on whether the switch-over date overlaps or not the query date range, one, or more, databases are needed and appropriately pushed into an array. Each element of `$sp_names` is a hash with keys SP_CNAME, SP_QNAME, DB_NAME, ARG_LIST and SQL_LIST. The first two keys are the stored procedure names for counting and querying, respectively, the database specified in the key DB_NAME. The last two keys, ARG_LIST and SQL_LIST, are arrays containing the argument list and corresponding SQL DB types of the stored procedures.

Assumption/Limitation: It is assumed that ARG_LIST and SQL_LIST are the same for both stored procedures in a given database. Generally this should not be too much of a problem since extra variables passed into a stored procedure can be ignored by the part of the stored procedure code that doesn't need those variables. If different ARG_LIST and SQL_LIST values are needed for each stored procedure, due to say extensive schema changes, then additional coding would be needed in the meta-data code & schema to handle that case.

5.0.2 getNumFound

The public method `getNumFound` loops over all the elements present in the structure `$sp_names` and calls in turn the stored procedure named in SP_CNAME to count the available records in the specified database for the instrument and date range being queried by the User. The Boolean variable `$per_year` is defined to be 1 for the ACOS database since it's schema uses a per-year table structure, otherwise it is 0. The method sums the counts returned by the stored procedure calls and returns the results. Also returned are the row counts on a per-year and per-igroup basis for later use.

5.0.3 getSPArgs

The public method `getSPArgs` takes as input the ARG_LIST and SQL_LIST keys in the structure `$sp_names` and merges the two arrays into an array of hashes, `$sp_args`, each element hash of which has two keys: QUERY_FIELD and SQL_TYPE. The method also counts the number of elements in ARG_LIST and generates a string of "?" placeholders duplicated the appropriate number of times for the given stored procedure. The method then returns the string containing the placeholders, together with the array of hashes `$sp_args`.

5.0.4 mapField

The public method `mapField` does any mapping necessary for a query that spans two databases whose individual schemas are different. The data in the VSO meta-data table `schema_mapping` is used to construct the field value and SQL type to be returned. If no mapping is needed for a given query field, that field is passed thru unchanged. The method returns the pair of variables, `$field` and `$sql_type` to the parent caller which then uses the pair in a call to the DBI method `bind_param` to bind the given variable value and it's SQL type before the final DBI method `execute` is called.

Note that for queries involving ACOS, the variable `$year` is passed in from the caller as an attribute to the parent object (`$self`), which is done on-the-fly depending on the query details. If other variables are likewise needed in the future, they too could be passed in as attributes, or the argument list of `mapField` could be generalized.

5.0.5 getRecords

The public method `getRecords` encapsulates the existing code that maps the data returned from the stored procedure into query fields to be displayed within an Apache instance, or piped to IDL, depending upon the source of the initial request. The reason for encapsulating the code into a method is that the existing logic of looping over the years present in the query needed generalization to use the data in the new structure `sp_names` since only data residing in ACOS needs to loop over the years present. Hence code is re-used depending upon whether the instruments being queried have data residing in ACOS, or not. The re-use

is accomplished by use of the value of `$igroup`, the instrument group no. Note that the *reference* to the key array `@records` is passed to `getRecords`, which pushes into the array in the parent the records found on any given call.

5.1 CHIP

The data for the CHIP instrument is split between two databases at HAO: ACOS and MLSO. The switch-over date was 11/06/2009. Data earlier than the switch-over date are located in the ACOS DB, while data after the switch-over date are located only in the MLSO DB.

The schema used in the ACOS DB has the data stored in separate tables on a per-year basis. The original VSO code in `MLSO.pm` had that schema hard-coded into the SQL statements used for counting and querying the data in the DB. The schema for the MLSO DB has all the subsequent data for CHIP after the switch-over date in 1 table `file`.

To simplify `MLSO.pm`, the ACOS-specific logic and schema related code was moved into the stored procedure `query_acos`:

```
query_acos => {
 script => qq{ CREATE PROCEDURE query_acos( YR INT,
                                            DT_START DATETIME,
                                            DT_END DATETIME,
                                            ACOS_INSTR VARCHAR(10),
                                            ACOS_TYPE VARCHAR(10) )
                    DETERMINISTIC READS SQL DATA
    BEGIN
     IF (YR = 2000) THEN
      SELECT datetime_obs+0 AS timestart,
        instrument AS instrument,
        wave_length AS wave,
        file_name AS fileid,
       concat_ws(': ', instrument, wave_length, processing, quality) AS info
      FROM tbl_2000
      WHERE (1=1)
      AND datetime_obs <= DT_START AND datetime_obs >= DT_END
          AND type = ACOS_TYPE AND instrument in (ACOS_INSTR);
     ELSEIF (YR = 2001) THEN
      SELECT datetime_obs+0 AS timestart,
        instrument AS instrument,
        wave_length AS wave,
        file_name AS fileid,
       concat_ws(': ', instrument, wave_length, processing, quality) AS info
      FROM tbl_2001
      WHERE (1=1)
      AND datetime_obs >= DT_START AND datetime_obs <= DT_END
          AND type = ACOS_TYPE AND instrument in (ACOS_INSTR);
     ELSEIF (YR = 2002) THEN
```

```
... skipping blocks for 2002 - 2012
      ELSEIF (YR = 2013) THEN
       SELECT datetime_obs+0 AS timestart,
          instrument AS instrument,
          wave_length AS wave,
          file_name AS fileid,
          concat_ws(': ', instrument, wave_length, processing, quality) AS info
       FROM tbl_2013
       WHERE (1=1)
       AND datetime_obs >= DT_START AND datetime_obs <= DT_END
            AND type = ACOS_TYPE AND instrument in (ACOS_INSTR);
      END IF;
     END;
  },
 },
```

where each year has it's own IF/ELSEIF block of code in the stored procedure. Once MySQL can handle dynamic table names in SQL, the stored procedure `query_acos` can be reduced considerably in length.

For the MLSO DB, the corresponding stored procedure `query_mlso` is :

```
query_mlso => {
 script => qq{ CREATE PROCEDURE query_mlso(
                                                DT_START DATETIME,
                                                  DT_END DATETIME,
                                                      MLSO_IGRP INT,
                                              MLSO_INSTR VARCHAR(10),
                                               MLSO_TELE VARCHAR(10),
                                                 MLSO_FT VARCHAR(10) )
                        DETERMINISTIC READS SQL DATA
     BEGIN
      DECLARE WAVELNGTH VARCHAR(10);
        SET WAVELNGTH = '10830';
        SELECT f.time_obs+0 AS timestart,
         MLSO_INSTR AS instrument,
         WAVELNGTH AS wave,
         f.filename AS fileid,
         concat_ws(': ', MLSO_INSTR, WAVELNGTH ) AS info
        FROM file AS f, filetype AS ft, instrume AS i, telescop AS t
        WHERE f.time_obs >= DT_START AND f.time_obs <= DT_END
        AND f.filetype = ft.ID AND ft.filetype = MLSO_FT
        AND f.instrume = i.ID AND i.INSTRUME = MLSO_INSTR
        AND f.telescop = t.ID AND t.TELESCOP in (MLSO_TELE);
      ELSEIF (MLSO_IGRP = 2) THEN
        SET WAVELNGTH = '7200';
        SELECT f.DATE_OBS+0 AS timestart,
```

```
            MLSO_INSTR AS instrument,
            WAVELNGTH AS wave,
            f.FILENAME AS fileid,
            concat_ws(': ', MLSO_INSTR, WAVELNGTH ) AS info
          FROM kcor_file AS f
          WHERE f.date_obs >= DT_START and f.date_end <= DT_END;
        END IF;
      END;
  },
},
```

where the local variable WAVELNGTH is set and returned since the schema in DB MLSO in tables `file` and `kcor_file` does not contain the wavelength of the instrument.

The Perl variable `$igroup`, representing the instrument group no. is mapped to the SQL variable MLSO_IGRP in the stored procedure `query_mlso`. The data in the above table is represented in `MLSO.pm` in a hash `$igroups` with the instrument label as the key. Hence for the case of the instrument CHIP, the IF block of code with `MLSO_IGRP` = 1 is executed in `query_mlso`.

5.2 K-Cor

The instrument K-Cor has an instrument group no. = 2, so the corresponding block of code that is executed in `query_mlso` is when `MLSO_IGRP` = 2.

5.3 CoMP

Under construction