Enhancements to SAS/Warehouse Administrator™

Terry Lewis, SAS Institute Inc., Cary, NC

ABSTRACT

This paper describes enhancements made to SAS/Warehouse Administrator software in the 1.2 release. Special attention is given to the Metadata Application Programming Interface. The Metadata API allows the Applications Programmer to create end user applications that are driven by metadata that is stored in the warehouse. Additionally, the Metadata API may be used to import third party metadata into the warehouse. Examples of API usage is explored and sample code fragments are discussed

PREREQUISITES

This paper assumes some previous knowledge of

- SAS/Warehouse Administrator 1.1 and the types of objects and metadata that reside in a data warehouse,
- Screen Control Language (SCL), a programming language used with SAS/AF® and SAS/FSP® software, and
- SCL applications using FRAME entries.

CONVENTIONS

In the code samples, SCL variables beginning with `i_` are instances of a class and variables beginning with `l_` are SCL lists.

INTRODUCTION

The primary emphasis in the 1.1 release of SAS/Warehouse Administrator was to deliver a user interface that would enable the site's Data Warehouse Administrator (DWA) to build and manage a data warehouse by defining metadata about how the data warehouse is structured and what is contained in the warehouse.

In contrast, the 1.2 release of SAS/Warehouse Administrator emphasizes the actual usage or exploitation of the metadata, and thus the data, contained in the warehouse. While the 1.1 release was targeted for use by the site's DWA, the primary audience for many of the new 1.2 features is the Applications Programmer who may wish to use the metadata as a driver to build applications for end users performing decision support.

The primary vehicle for this metadata exploitation is the **Metadata Application Programming Interface,** or **Metadata API**. This paper will explain the structure of the Metadata API, how it works, and how it can be used to develop applications.

Another application-related feature in release 1.2 that will be discussed briefly is the **Add-Ins** facility. This facility, formerly known as "User-Written Tools", allows you to plug applications directly in to the SAS/Warehouse Administrator user interface. This can be useful for integrating applications that are developed for use by the DWA, rather than end users.

In addition to the Metadata API and the Add-Ins facility, release 1.2 contained several Job Scheduler enhancements. The Job Scheduler is in beta status and discussion of those enhancements is beyond the scope of this paper.

REQUIREMENTS

SAS/Warehouse Administrator software runs on any platform running SAS 6.12 or above, typically a PC running Windows, and requires Base SAS® software and SAS/FSP® software. The generated code from SAS/Warehouse Administrator can run on any SAS 6.08 platform or above. Additional SAS products may be required depending on what warehouse facilities are used. For example, SAS/Connect® software would be required for remote processing and SAS/Access® software would be required for accessing or loading data stored in DBMS tables.

Applications written using the Metadata API require SAS/Warehouse Administrator software, release 1.2 or above, Base SAS software, release 6.12 or above, and SAS/AF software, release 6.12 or above.

METADATA-DRIVEN APPLICATIONS

Metadata is the information internal to an application that describes elements in that application. In SAS/Warehouse Administrator, metadata is created as the DWA defines, via the SAS/Warehouse Administrator user interface, the warehouse environment, the data warehouses, and the objects that are contained within the environment and warehouses, such as tables, infomarts, mappings, etc.

There are different kinds of metadata stored in SAS/Warehouse Administrator. Some metadata is of a business or informational nature, for example, what business subjects the warehouse is organized in. Other metadata is more technical, such as the columns in a table or how data is transformed before loading it in the warehouse.

Typically, once the data warehouse metadata for one or more business subjects is defined and the associated warehouse tables are populated with live data, the DWA will pass the information about the metadata to the Applications Programmer who will begin to design applications for end users. Most applications written for end users using warehouse data are of an analytical nature, used in support of making business decisions.

When the Applications Programmer starts to design their application, the Metadata API can assist them by allowing them direct access into the metadata that the DWA created without having to go through the user interface. Rather than hard-coding table names, locations, etc., the application can become **metadata-driven**, i.e., all discrete values used by the application are read from the metadata via the Metadata API. If designed properly, a Metadata API application can be insulated from any changes to the data warehouse that occur over time.

Decision support applications as described above will normally access the metadata in a read-only manner. However, the Metadata API can also be used to develop applications that will actually write to the metadata. Usually, these types of applications are more suitable for use by the DWA rather than the end user since usually only the DWA has the authority to update the metadata. For example, an application could be written to mass import hundreds or thousands of table definitions into the

warehouse, rather than having to laboriously define each table via the user interface.

METADATA REPOSITORIES

Before you can understand the Metadata API, you must understand how SAS/Warehouse Administrator stores its metadata.

Metadata is stored in a **Metadata Repository**. A metadata repository may be *standalone*, such as the SAS/EIS® metabase, or may be *partitioned*. In a partitioned structure, you have one or more *primary* repositories, each of which may have one or more *secondary* repositories associated with it. Partitioning allows different kinds of metadata to be stored in different locations or formats. SAS/Warehouse Administrator uses a partitioned metadata repository, where the primary repository contains common, global metadata that is shared by all data warehouses. Each secondary repository that is associated with the primary repository stores metadata that is particular to a single data warehouse. In the SAS/Warehouse Administrator user interface, the primary repository is created when a Data Warehouse Environment is defined in the Data Warehouse Environment Properties Window:

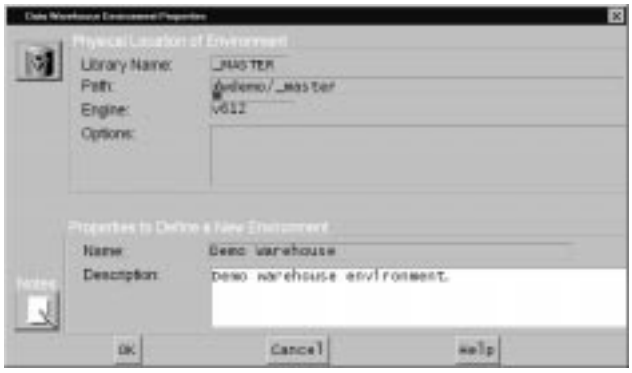

Figure 1 - Data Warehouse Environment Properties Window

The primary repository is always assigned the libname _MASTER when accessed by SAS/Warehouse Administrator or a Metadata API application.

A secondary repository is created for each data warehouse when a warehouse is defined in the Data Warehouse Properties Window as shown in figure 2:

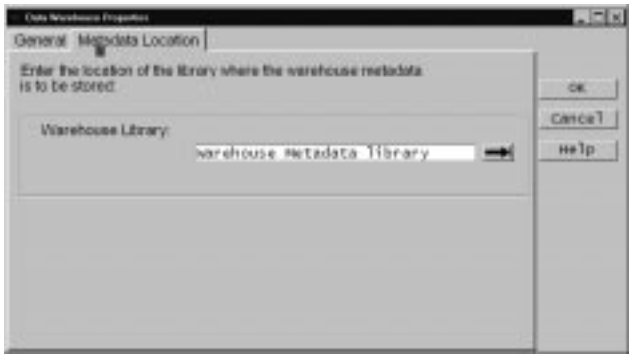

Figure 2 - Data Warehouse Properties Window

Only one single secondary repository may be active at one time and is assigned the libname of _DWMD when accessed by SAS/Warehouse Administrator or a Metadata API application.

Currently, SAS/Warehouse Administrator metadata repositories are physically stored as a series of SAS data sets. However, the actual physical format of the metadata may change to some other type of storage in the future. Using the Metadata API to access metadata repositories insulates the application from any potential changes in format since the Metadata API itself is designed so that any physical storage changes will not change the information that is delivered to the application. **You should always go through the Metadata API to access the Metadata Repositories -- never read or write to the repositories directly!**

METADATA OBJECTS

Metadata repositories contain metadata objects. Each metadata object in a repository has **properties**, which are the attributes of the object. For example, a column in a Warehouse Detail Table has properties like name, description, length, type (character or numeric), format, informat, etc.

The name and description are properties that appear in all metadata objects and are usually assigned by the DWA when defining the object through the SAS/Warehouse Administrator user interface. An additional property that is common to all objects is the ID, also known as the **Metadata Identifier.** The Metadata Identifier uniquely identifies the metadata object and is described in the form:

reposid.typeid.instanceid

reposid is an 8-character identifier assigned to a particular warehouse repository when the repository was created. This value is returned to the application when a connection is made to the repository.

typeid is an 8-character identifier that is the **Metadata Type** of the object being identified. SAS/Warehouse Administrator supports many different kinds of warehouse objects and those objects are categorized into specific Metadata Types. For example, WHDETAIL is the Metadata Type for a Warehouse Detail Table. WHCOLODD identifies an object as being of an ODD (Operational Data Definition) Column Type. All Metadata Types are defined in Appendix 1 in *SAS/Warehouse Administrator Metadata API Reference, Release 1.2.*

Instanceid is an 8 character unique identifier that distinguishes one metadata object from all other objects of the same type and is assigned when the object is created.

METADATA API BASICS

The Metadata API is SCL-based, i.e., the interface is implemented via a single SAS/AF class and several method calls. The Metadata API method calls are embedded within the application that is reading or writing the metadata, known as the *client application.*

The **METAAPI** class is the class that is used to communicate with the Metadata API to read and write metadata, i.e., all read and write methods are sent to an instance of the METAAPI class. Using the METAAPI class is a simple matter of instantiating it so methods can be sent to it:

```
i_api = instance(loadclass
```

```
('SASHELP.METAAPI.METAAPI.CLASS'));
```

Once instantiated, various Metadata API methods can be sent to the instanced class via CALL SEND, for example:

```
call send(i_api, '_GET_METADATA_',
                  l_rc, l_meta);
```

Each Metadata API method has a documented parameter format. In the method illustrated above, the _GET_METADATA_ method is sent to an instance of the METAAPI class to retrieve information about a particular metadata object. The third parameter, `l_rc`, is an SCL list that will contain return code information from the method call and the fourth parameter, `l_meta`, is known as the **Metadata Property List.**

The Metadata Property List is the primary mechanism by which information is exchanged for most Metadata API methods. When a Metadata API method is called, the Metadata Property List contains information the method needs as input, such as the Metadata Identifier of the object in question. Upon return, the Metadata API will have updated the Metadata Property List to contain the information you have requested.

For example, if you wanted to request additional information about a particular Detail Table in a Data Warehouse, you could prepare the Metadata Property List like so:

```
l_meta = makelist();
l_rc = makelist();

id = 'A000000E.WHDETAIL.A000002X';
l_meta = insertc(l_meta, id, -1,
         'ID');
l_meta = insertc(l_meta, ' ', -1,
         'TABLE NAME');
l_lib = makelist();
l_meta = insertl(l_meta, l_lib, -1,
         'LIBRARY');
l_cols = makelist();
l_meta = insertl(l_meta, l_cols, -1,
         'COLUMNS');

call send(i_api, '_GET_METADATA_',
                  l_rc, l_meta);
```

If you displayed the Metadata Property List, `l_meta`, prior to the _GET_METADATA_ method call, it would look like:

```
L_META=
 ( ID='A000000E.WHDETAIL.A000002X'
  TABLE NAME=' '
  LIBRARY=()[53]
  COLUMNS=()[55]
 )[49]
```

The Metadata Property List in this example uses the ID parameter to identify the metadata object. Note that *typeid* is set to WHDETAIL which maps to the Warehouse Detail Table type. The additional empty parameters, TABLE NAME, LIBRARY, and COLUMNS, instruct the Metadata API to fill in those specific properties with information for that object.

After the _GET_METADATA_ method executes successfully, the Metadata Property List would contain information about the properties you requested, for example:

```
L_META=
( ID='A000000E.WHDETAIL.A000002X'
  TABLE NAME='PRODUCT'
  LIBRARY=( ID='A0000001.WHLIBRY.A000000U'
            NAME='Warehouse Data Library'
            DESC=''
          )[53]
  COLUMNS=( (ID='A000000E.WHCOLDTL.A0000032'
              NAME='PRODNUM'
              DESC='product number'
             )[849]
            (ID='A000000E.WHCOLDTL.A0000034'
              NAME='PRODNAME'
              DESC='product name'
             )[957]
            (ID='A000000E.WHCOLDTL.A0000036'
              NAME='PRODID'
              DESC='product id/abbreviation'
             )[933]
            (ID='A000000E.WHCOLTIM.A00000FU'
              NAME='_LOADTM'
              DESC='DateTime Stamp of
                    when row was loaded'
             )[935]
          )[55]
 )[49]
```

The LIBRARY property specifies which library the table resides in and the COLUMNS property enumerates the columns in the table. Further information can be obtained, via the _GET_METADATA_ method, about each of those objects since a Metadata Identifier is returned for each one.

You may have noticed in the COLUMNS property SCL list that two different types of objects were returned -- three WHCOLDTL objects and one WHCOLTIM object. The WHCOLDTL type is a Detail Column and the WHCOLTIM type is a Detail Table Time Column. Both the WHCOLDTL and the WHCOLTIM objects are **subtypes** of their parent, the WHCOLUMN type. Subtypes allow better definition and granularity of the roles that metadata objects perform. In our example, the WHCOLTIM type identifies that column as being a special column that is used to hold the load time stamp of the table. Subtypes may inherit most or all of their properties from their parent type. Each metadata type has specific properties that it will accept as input and subsequently return information about. Types and their properties are fully documented in the *SAS/Warehouse Administrator Metadata API Reference.*

Note that the purpose of the previous example was to illustrate how the Metadata Property List is used. Normally, you should not assign the literal value of the Metadata Identifier in your application code. If the particular metadata object associated with that Metadata Identifier was deleted from the metadata repository and subsequently re-added, its Metadata Identifier would change which would invalidate the hard-coded identifier and cause program errors. There are other methods, described later in this paper, that enable you to retrieve the Metadata Identifier for an object programmatically.

USING THE METADATA API

Note that all Metadata API methods are documented in the *SAS/Warehouse Administrator Metadata API Reference.* A few of the most commonly used methods are used in the examples below.

ATTACHING A PRIMARY REPOSITORY

Before you can read or write metadata, you must first attach to the metadata repositories that contain the metadata. This is usually one of the first things performed in a Metadata API application program. In a partitioned metadata repository structure, the _SET_PRIMARY_REPOSITORY_ method is used to attach to the primary metadata repository. In SAS/Warehouse Administrator, the primary repository contains global metadata, like hosts, libraries, contacts, ODDs, etc., that are shared between all warehouses.

In the example below, the only item in the Metadata Property List is the LIBRARY parameter, which contains the information about where the repository resides. Additionally, we set the `repostype` parameter to WHDWENV, since, in SAS/Warehouse Administrator, the primary metadata repository is a Data Warehouse Environment and its Metadata Type is WHDWENV. In the following example running on Windows, a primary repository is attached:

```
path = '!SASFOLDER\whouse\dwdemo\_master';
i_api = instance(loadclass
        ('SASHELP.METAAPI.METAAPI.CLASS'));
repostype = 'WHDWENV';

l_meta = makelist();
l_meta_returned = makelist();
l_lib  = makelist();
l_meta = insertl(l_meta, l_lib, -1,
         'LIBRARY');
l_path = makelist();
l_lib  = insertl(l_lib, l_path, -1,
         'PATH');
l_path=insertc(l_path, path, -1);

call send(i_api,'_SET_PRIMARY_REPOSITORY_',
          rc, l_meta, repostype,
          primary_repos_id, l_meta_returned);
```

On return, the Metadata Property list `l_meta_returned` looks like

```
L_META_RETURNED=
( ID='A0000001.WHDWENV.A0000001'
  NAME='Demo Warehouse'
  DESC='Demo warehouse environment.'
 )[49]
```

where ID is the Metadata Identifier of the Data Warehouse Environment object. The SCL variable `primary_repos_id` is set to `A0000001`. Additionally, issuance of the _SET_PRIMARY_REPOSITORY_ method results in the _MASTER libname being allocated to your SAS session.

GETTING THE LIST OF SECONDARY REPOSITORIES

The _GET_METADATA_ method is the primary method used to read metadata in a metadata repository. In the example shown under the previous topic *Metadata API Basics*, the _GET_METADATA_ method was used to retrieve LIBRARY and COLUMN information about a Warehouse Detail Table.

The _GET_METADATA_ method can also be used to retrieve a list of secondary repositories linked to a primary metadata repository, since, in SAS/Warehouse Administrator, the list of secondary repositories (metadata type WHDW) is a property of the primary metadata repository (metadata type WHDWENV) via the REPOSITORIES property.

Continuing our previous example, we can reuse the returned property list, `l_meta_returned`, as the Metadata Property List for the _GET_METADATA_ method:

```
l_reps = makelist();
l_meta = setniteml(l_meta_returned, l_reps,
         'REPOSITORIES');
call send(i_api, '_GET_METADATA_',
         l_rc, l_meta_returned);
```

Successful execution of this method returns the list of secondary repositories that are associated with the primary metadata repository:

```
L_META_RETURNED=
( ID='A0000001.WHDWENV.A0000001'
  NAME='Demo Warehouse'
  DESC='Demo warehouse environment.'
  REPOSITORIES=( (
           ID='A0000001.WHDW.A000000E'
           NAME='Corporate'
           DESC='Demonstration
warehouse.  Note that all information
contained within was created for
demonstration purposes only.'
                 )[693]
                (
           ID='A0000001.WHDW.A000007M'
           NAME='Test'
           DESC='The Test warehouse is
not populated but is included here to
show how the SAS/Warehouse Administrator
can help you build and manage multiple
Data Warehouses.'
                )[699]
               )[129]
 )[49]
```

In this example, the `l_meta_returned` property list contains property information on two secondary repositories linked to the primary repository.

The _GET_METADATA_ method also contains optional parameters that control how much metadata is returned in the Metadata Property List. For example, if the ALL parameter is activated, the Metadata Property List will return all properties of the metadata object, not just NAME, DESC, and ID. Note that specification of these optional parameters may result in slower performance.

ATTACHING A SECONDARY REPOSITORY

The _SET_SECONDARY_REPOSITORY_ method is used to attach to any secondary metadata repositories so that subsequent read or write methods can be issued to retrieve metadata pertaining to a specific warehouse. Only one secondary repository can be active at a time.

Continuing our previous example, we issue the _SET_SECONDARY_REPOSITORY_ method after the list of secondary repositories has been returned by the _GET_METADATA_ method:

```
/* just point to the first warehouse */
l_rep = getiteml(l_reps, 1);
dw_id = getnitemc(l_rep, 'ID');

l_smeta = makelist();
```

```
l_smeta = insertc(l_smeta, dw_id, -1, 'ID');
call send(i_api,'_SET_SECONDARY_REPOSITORY_',
          rc, l_smeta, secondary_repos_id);
```

Upon return, the SCL variable `secondary_repos_id` will contain the id of the secondary repository and the _DWMD libname will be allocated to your SAS session.

RETRIEVING A LIST OF METADATA OBJECTS

As mentioned earlier, you should avoid "hard-coding" explicit Metadata Identifiers in your Metadata API application program. However, many Metadata API methods, such as _GET_METADATA_, require a Metadata Identifier. One method of retrieving Metadata Identifiers is to use the _GET_METADATA_OBJECTS_ method. This method allows you to retrieve Metadata Identifiers for objects of the same type. For example, if you wished to present a pop-up menu of all Detail Tables in a particular warehouse, the following code could be used:

```
l_meta_returned = makelist();
l_rc = makelist();

type = primary_repos_id || '.WHDETAIL';
call send(i_api,'_GET_METADATA_OBJECTS_',
          l_rc, type,l_meta_returned);

if l_rc =  0 then
   idx = popmenu(l_meta_returned);
```

The list of objects looks like

```
L_META_RETURNED=
( A000000E.WHDETAIL.A000001L='Customer detail
  table'
  A000000E.WHDETAIL.A000002X='Product detail
  table'
  A000000E.WHDETAIL.A000003M='Customer detail
  table'
  A000000E.WHDETAIL.A000004H='Sales fact
  table'
  A000000E.WHDETAIL.A000005U='Oracle'
  A000000E.WHDETAIL.A000006Q='Sybase'
  A000000E.WHDETAIL.A000007L='Remote Detail
  Table'
  A000000E.WHDETAIL.A000008I='Suppliers'
 )[49]
```

and the menu displayed to the user appears as

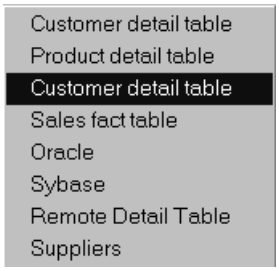

Figure 3 - Pop-up Menu

When the end user makes a selection from the pop-up menu, the Metadata Identifier can be retrieved via the `nameitem()` SCL function and can then be used in subsequent Metadata API methods.

READING METADATA

As discussed previously, the _GET_METADATA_ method is used to read metadata after the metadata repositories have been attached. In previous examples, the _GET_METADATA_ method was used to retrieve additional property information about table and repository metadata objects.

The next example demonstrates how to retrieve **process metadata properties.** Fully defined detail tables in SAS/Warehouse Administrator have a process associated with the table that dictates how the data is extracted from the operational systems (ODDs), transformed, and loaded into the target table. This process is depicted visually in the Process Editor, for example:

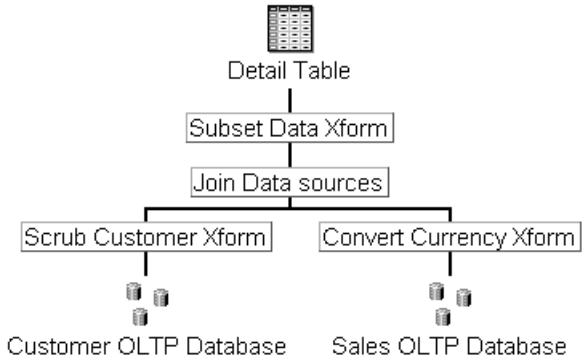

Figure 4 - Process Flow of a Warehouse Table

In figure 4, the process flow depicts operational data being extracted from two operational sources, *Customer OLTP Database* and *Sales OLTP Database*. The data is then passed through two transformations, *Scrub Customer Transformation* and *Convert Currency Transformation*, before being joined by the *Join Data Sources* mapping. The data passes through one final transformation, *Subset Data Transformation*, before being loaded into the table.

In our sample Metadata API application, we would like to build the list of transformations that the data is passing through, as well as determine the location of the transformation source code.

Assuming that we already have the Metadata Identifier of the desired table, we use the INPUT SOURCES property to retrieve the "closest" input sources of the detail table. We then call a separate SCL program, GETSRCS.SCL, to recursively retrieve the rest of the input sources and place them in an SCL list. Note that error-checking and other housekeeping code has been removed for brevity:

```
id = 'A0000002.WHDETAIL.A0000005';
l_meta    = makelist();
l_sources = makelist();
l_xforms  = makelist();
l_rc      = makelist();

/*
 * Get the closest input sources for
 * the detail table.
 */
```

```
l_meta = insertc(l_meta, id, -1,
         'ID');
l_meta = insertl(l_meta, l_sources, -1,
         'INPUT SOURCES');
call send(i_api, '_GET_METADATA_',
                  l_rc, l_meta);

/*
 * Now call the recursive routine
 */
call method('GETSRCS.SCL', 'GETSRCS', l_rc,
            i_api, l_sources, l_xforms);
```

GETSRCS.SCL is a recursive routine that calls itself for each input source:

```
GETSRCS: method l_rc 8 i_api 8 l_sources 8
                l_xforms 8;

   num_sources=listlen(l_sources);
   if num_sources > 0 then do;

     /*
      *  A process object may have more than
      *  one source, so do them in a loop
      */
     do i = 1 to num_sources while (l_rc=0);

       l_source = getiteml(l_sources, i);
       input_name = getnitemc(l_source,
                    'NAME');
       input_id = getnitemc(l_source,
                    'ID');
       input_type = scan(input_id, 2, '.');

       /*
        *  Only retrieve process objects,
        *  not "icon" objects
        */
       call send(i_api, '_IS_SUBTYPE_OF_',
                 l_rc,input_type,'WHTBLPRC',
                 is_process_table);

       if is_process_table then do;

         /*
          *  The next statement inserts
          *  the transformation in the
          *  l_xforms list
          */
         l_xforms = insertc(l_xforms,
                    input_name,-1,input_id);

         /*
          *  Get the closest input sources
          *  for the current process object
          */
         l_next_sources = makelist();
         l_source = setniteml(l_source,
                    l_next_sources,
                    'INPUT SOURCES');

         call send(i_api, '_GET_METADATA_',
                   l_rc, l_source);

         /*
          *  Call this routine recursively
          *  for next set of input sources.
          */
         num_srcs = listlen(l_next_sources);
         if num_srcs > 0 then
           call method('GETSRCS.SCL',
                       'GETSRCS', l_rc,
                       i_api, l_next_sources,
                       l_xforms);
       end;

     end;

   end;

endmethod;
```

After the original method call to GETSRCS.SCL in the main program is complete, the `l_xforms` list contains the list of transformations that preceded the load of the target detail table:

```
L_XFORMS=
( A0000002.WHTBLUSR.A0000016=
    'Subset Data Xform'
  A0000002.WHTBLMAP.A000000G=
    'Join Data sources'
  A0000002.WHTBLUSR.A000000K=
    'Scrub Customer Xform'
  A0000002.WHTBLUSR.A000000N=
    'Convert Currency Xform'
 )[53]
```

In this format, the list can easily be displayed in a pop-up menu or listbox. A few more method calls and the location information could be obtained from the metadata:

```
do i = 1 to listlen(l_xforms);

    l_row = makelist();
    rc = clearlist(l_meta, 'Y');

    /*
     *  Get the process associated with
     *  this transformation.
     */
    l_meta=insertc(l_meta,
                   nameitem(l_xforms,i),
                   -1, 'ID');
    l_meta = insertl(l_meta, l_process,
                   -1, 'PROCESS');
    call send(i_api, '_GET_METADATA_',
                     l_rc, l_meta);
    /*
     *  Now get the source file associated
     *  with the process.
     */
    l_process = insertl(l_process, l_srcfil,
                        -1, 'SOURCE FILE');
    call send(i_api, '_GET_METADATA_',
                     l_rc, l_process, 0, 1);
    /*
     *  The FULL ENTRY property gives us the
     *  location of the transformation.
     *  Insert that into the current row.
     */
    src_file = getnitemc(l_srcfil,
                 'FULL ENTRY');
    l_row = insertc(l_row,
                    getitemc(l_xforms,i),
                    -1, 'TRANSFORMATION');
    l_row = insertc(l_row, src_file,
                    -1, 'LOCATION');

    /*
     *  Insert the current row into the
     *  full table.
     */
    l_data = insertl(l_data, l_row, -1);

end;
```

The l_data SCL list can then be easily attached to a data table:

```
call notify('.', '_GET_WIDGET_', 'W_TABLE',
                 w_table);
i_listm = instance(loadclass
            ('sashelp.fsp.list_m.class'));
call send(i_listm, '_SET_DATA_LIST_',
            l_data);
call notify('w_table', '_ATTACH_',
             i_listm);
```

resulting in:

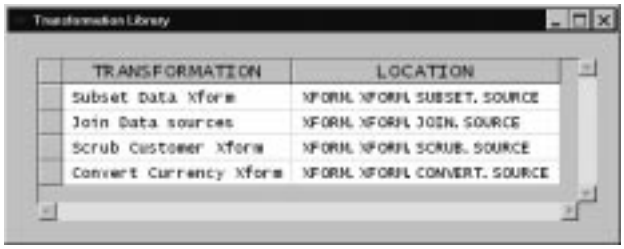

Figure 5 - Transformation Data Table

From this data table, it would be an easy task to enable the selective browsing of either a transformation description, NOTES on the transformation, or even the actual source code of the transformation itself. With more work, one could even see this example evolving into a Transformation Library Management tool. This demonstrates that the Metadata API can be used to actually extend the feature set of SAS/Warehouse Administrator to provide other "views" of the metadata that aren't currently supplied by the default software.

WRITING METADATA

While the Metadata API is primarily used by applications programs for reading metadata, three methods are available that access the metadata in a write mode. _ADD_METADATA_ is used to add new metadata objects, _UPDATE_METADATA_ is used to update existing metadata objects, and _DELETE_METADATA_ is used to delete metadata objects.

In SAS/Warehouse Administrator, you can read all metadata objects with _GET_METADATA_, however, you can only write to certain objects. The *SAS/Warehouse Administrator Metadata API Reference* documents what metadata objects can be added, updated, or deleted, however, as a general rule, the objects that are displayed in the Explorer window can be written to. Additionally, the global objects displayed in the Setup window can also be written to. The information displayed in the Process Editor window, i.e., process-related information, for the most part cannot be written to.

The primary purpose of the write facility is to be able to import metadata into a SAS/Warehouse Administrator metadata repository without having to go through the user interface. If you had to define hundreds of tables in your warehouse, it would be far quicker and easier to write a simple metadata import program to define those tables. An import program would permit the DWA to read an external data source, perhaps derived from a warehouse modeling tool, and write that information to the metadata repository programmatically.

The following example is a simple example of defining a new subject in a data warehouse:

```
l_meta=makelist();

/*
 *  Set the group that will contain the
 *  new subject
 */
l_groups=makelist();
l_group=makelist();
l_groups=insertl(l_groups,l_group,-1);
l_group=insertc(l_group,wh_group_id,-1,'ID');
l_meta=insertl(l_meta,l_groups,-1,'GROUP');

/*
 *  Set up the property list.  A blank ID
 *  implies an ADD, rather than UPDATE.
 */
new_type = repos_id || '.WHSUBJCT';
l_meta=insertc(l_meta,new_type,-1,'ID');
l_meta=insertc(l_meta,'NEW SUBJECT',-
               1,'NAME');
l_meta=insertc(l_meta,
            'New subject added through API',
             -1,'DESC');

/*
 *  Add the subject
 */
call send(i_api,'_ADD_METADATA_',
          l_rc, l_meta);
```

When you invoke the SAS/Warehouse Administrator Explorer, the new subject appears in the directory as shown in figure 6:

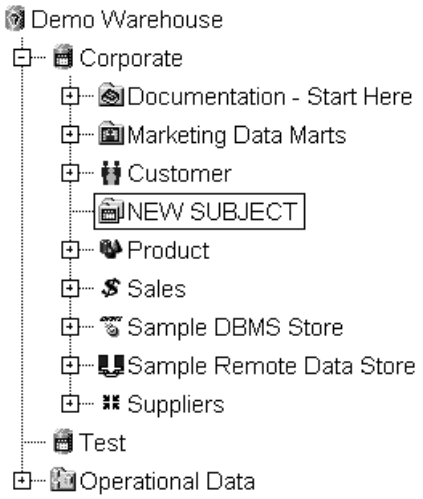

Figure 6 - Adding a New Subject

While this is a very simple example, it demonstrates how easy it is to use the Metadata API to update metadata in a write mode. Metadata properties, metadata types, list formats, etc. are consistent no matter what methods are being employed.

THE ADD-INS FACILITY

Release 1.1 of SAS/Warehouse Administrator had a facility known as "User Written Tools" that allowed an Applications Programmer to insert their own applications in the SAS/Warehouse

Administrator user interface. Those applications were accessible from a pull-down menu.

With Release 1.2, this facility has been enhanced to support Metadata API applications and has been renamed to "Add-Ins". Sample Metadata API applications, with associated source code, have been shipped with release 1.2 to demonstrate how this facility works.

The applications available from the Add-Ins pull-down menu are typically applications that are targeted more at the DWA, rather than the end user, since they hook into the SAS/Warehouse Administrator user interface itself. This facility allows you to extend the functionality of SAS/Warehouse Administrator itself, i.e., if SAS/Warehouse Administrator doesn't contain a desired feature, you can write an application yourself to provide that feature.

The Add-Ins facility is driven by the Add-In Tool Registry. The Add-In Tool Registry determines what Add-In applications are presented in the pull-down menu, where they are located, what parameters the application expects, and other interface considerations. The default Add-In Tool Registry is a SAS dataset called **WATOOLS** and resides in the SASHELP library. It may also reside in a library pointed to by the **_SASWA** libname. If the _SASWA.WATOOLS is present, it's entries and not those of SASHELP.WATOOLS will be used. This allows you to test your Add-In application without making changes to the default or production copy of WATOOLS. The format of the Tool Registry dataset is documented in the README.TXT file on your installation media.

In the coming year, more Add-In applications will be released from the Institute that will further demonstrate how this facility can be used.

CONCLUSION

The Metadata API and the Add-Ins facility in Release 1.2 of SAS/Warehouse Administrator significantly enhance the functionality of the software and allow the Applications Programmer to begin leveraging their site's investment in metadata. Additionally, the Metadata API can be used to extend the software beyond its current capabilities. This further strengthens the power and flexibility of SAS/Warehouse Administrator to build, manage, and exploit data warehouses.

REFERENCES

Lewis, Terry, SAS Institute Inc. (1997), "*SAS/Warehouse Administrator Usage and Enhancements",* Proceedings of the Twenty-Second Annual SAS Users Group International Conference

SAS Institute Inc. (1997), *SAS/Warehouse Administrator Metadata API Reference, Release 1.2,* Cary, NC: SAS Institute Inc.

SAS Institute Inc. (1996), *SAS/Warehouse Administrator User's Guide,* Cary, NC: SAS Institute Inc.

AUTHOR CONTACT

Terry Lewis, SAS Institute Inc., 100 SAS Campus Dr., Cary, NC 27513, (919)677-8000, ext. 7778, email snotll@wnt.sas.com