Paper 287-2010

An automated security framework: A metadata approach

Jorge Jordá Morlán, Lloyds Banking Group, UK

ABSTRACT

Do you need to update your data warehouse (tables, views, cubes) during the day time but need to ensure no users are using it? Do you currently need to do manual checks prior to the update, which can be time consuming and, sometimes, inaccurate? This paper proposes an automated solution so you don't need to worry about all those additional checks. Also, it incorporates a control panel stored process to provide an easier administration.

The solution utilizes the SAS Open Metadata Interface to programmatically communicate with the SAS Metadata Server and control the status of the different objects we want to manage.

INTRODUCTION

In the past, if you ran a job that updates a table during working hours, you had to previously manually check that nobody was using it and then remove access in SAS Management Console. If the job had to update several tables then the risk of doing something wrong or miss one object increases. In the best case you would have an Access Control Template to remove access to all objects at the same time, but if you have several groups of users it gets more and more complex.

Also, the fact that you have to manually turn the permissions off meant you couldn't automate the job 100%.

The following figure shows a normal process of updating a resource:

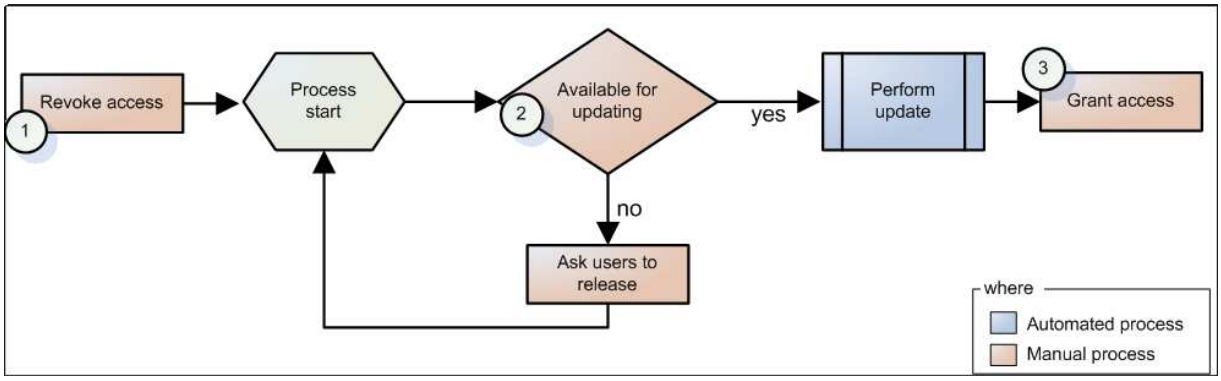


Figure 1: Common manually controlled updating process

As we can see, there are three main sections that force us to have a manual process:

1. Manually check nobody is using a resource.
2. Ensuring nobody will access it while updating the table.
3. Granting back access to the users once the update is completed (for this, you need to keep an eye to the process so that when it finishes you can grant access).

Only if we solve each of the individual problems we will be able to fully automate our process. The following three sections explain our approach of performing each of these tasks.
An additional section covers the integration of the three methods in a flow and the last one presents conclusions and benefits of using this approach.

The following figure shows how the new process will look once we have automated all parts:

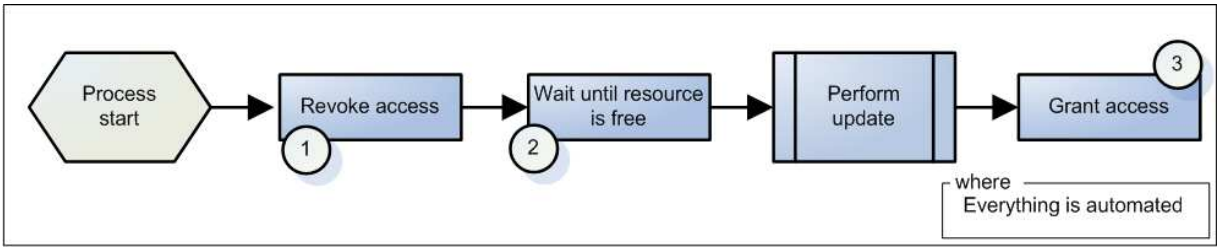


Figure 2: Automated updating process

DATA SCENARIO

The solution has been implemented to work with several data schemas; each schema refers to a business subject. In this paper we will be covering only two schemas, as an example: O4 (BM Applications) and F215 (MSP Performance ME). Each schema consists of a number of objects registered in the metadata (mostly tables and cubes). There is a custom ACT for each schema that groups all the objects together:

Schema: BM Applications
ACT: BM Applications (O4)
Tables:
- **Library:** DETAIL
  - BMS_APPLICATION_DIM
  - BMS_CASE_DIM
  - BMS_EMPLOYMENT_DIM
  - BMS_VALUATION_DIM

Schema: MSP Performance ME
ACT: F215
Tables:
- **Library:** DETAIL
  - BTH_PERFORMANCE_ME_DIM
  - BTH_PERFORMANCE_ME_FACT
- **Library:** MART
  - BTH_PERFORMANCE_ME_MART

Both Detail and Mart libraries are defined in the SMC as SPDE libraries although the propositions in this paper should be valid to any kind of file based library.
Each custom ACT is used for grouping purposes and doesn't contain any access control entries for any user or group. This could also have been achieved by having a lookup table with the pair: SCHEMA, OBJECT.

SOLVING THE PROBLEMS

In the following sections we will be concentrating in each of the nodes of the problems. Once we have solved all, our flow will be ready to be fully automated.

1. REVOKING ACCESS TO AN OBJECT

In order to remove the access to tables or cubes there are several approaches we can follow, but we can classify them in two types:

- Manual: This approach will need the administrator to log into SMC and manually remove access in the object or ACT. Although this solution does work, it is not ideal for automated environments as requires manual intervention.
- Automatic: This will use SAS code to automatically grant or remove the permissions.

For our case, we will explore the automatic solution by using the SAS Metadata facility. A special helper ACT is required for this task: The deny ACT (alias deny to everybody). If applied, this ACT will deny all permissions to the desired users and groups (as per the ACT definition). The idea of this is to automatically apply the deny ACT to the desired objects:

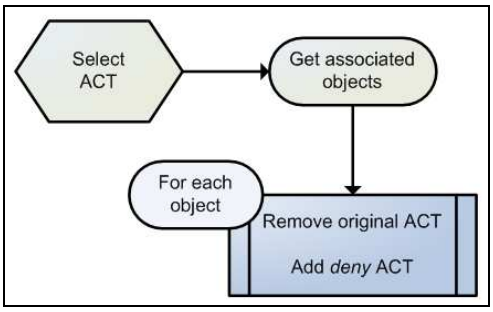


Figure 3: Removing access

The following steps are needed to complete the flow:

1. Get the ACT associated to the schema we want to analyse. This is straight forward as the ACT and the schema name are the same.
2. Get the list of associated objects, using the following metadata query[1].

```
<GetMetadataObjects>
  <Reposid>repository_id</Reposid>
  <Type>AccessControlTemplate</Type>
  <Objects/>
  <NS>SAS</NS>
  <Flags>2309</Flags>
  <Options>
    <Templates>
      <AccessControlTemplate name="" id=""/>
      <PhysicalTable name="" id="" />
    </Templates>
    <XMLSelect search="AccessControlTemplate[@Name='MyACT']"/>
   </Options>
</GetMetadataObjects>
```

This would return table objects, for cubes we would have to add in the Templates tag:

```
<Cube name="" id="" />
```

Note that this code would search for an ACT called MyACT.
The output of the query[2] will provide a dataset containing the table name and table ID. We need the Table ID to find its properties in the metadata.

[1] Appendix 1 shows how to perform xml metadata query as well as how to read the result.
[2] Prior to having the dataset output, an additional processing is required. See appendix 1.

3. For each table from the previous step we need to remove the existing ACT and add the deny one. For this we will again use a metadata query:

```
<UpdateMetadata>
   <Metadata>
   <AccessControlTemplate id="actId" >
       <Objects function="Remove">
           <PhysicalTable ObjRef="tableId"/>
       </Objects>
   </AccessControlTemplate >
   </Metadata>
   <NS>SAS</NS>
   <Flags>268435456</Flags>
   <Options/>
</UpdateMetadata>
```

Removes the relation between the ACT and the Table.

Note that for the query, we need an additional element: `actId`. This is the metadata ID for the ACT we want to remove. We can search for it using the following code:

```
<GetMetadataObjects>
   <ReposId>repositoryId</ReposId>
   <Type>AccessControlTemplate</Type>
   <Objects/>
       <NS>SAS</NS>
       <Flags>2436</Flags>
   <Options>
       <XMLSelect search="AccessControlTemplate[@Name='F215']"/>
   </Options>
</GetMetadataObjects>
```

This would cover the removal of the ACT. For the addition of the deny one, we would repeat the same operations but replacing the Remove value in the function attribute to Append, and obviously search for the appropriate deny ACT.

Once the deny ACT has been applied, the permissions of each object will look like the following table, so nobody could access it:

Permissions	Grant	Deny
ReadMetadata	☐	☑
CheckInMetadata	☐	☑
Create	☐	☑
Administer	☐	☑
WriteMetadata	☐	☑
Read	☐	☑
Write	☐	☑
Delete	☐	☑

Figure 4: Table permissions after applying the deny ACT.

2. WAIT UNTIL A RESOURCE IS FREE

We need to ensure that all the elements in a schema are free and available to update.
There are several ways of checking if a resource is currently being used. We have classified them in two groups:

- Intrusive: You need to actually "modify" the resource to check if it's available. Normally you would do a dummy modification. An example of checking if a table is available could be:

```
proc datasets lib=detail;
       modify my_table_dim (label="my_table_dim");
quit;
```

This code would work but it is dangerous to actually modify production tables, even if you know what you are doing. If the resource is not available, the SAS log would show an error. Errors or warnings can appear in the log.

- Non intrusive: It uses other mechanisms at the operating system level, where you can check if a physical file is currently being used. No errors or warnings are produced in the SAS log.

  Where possible, it is recommended the usage of non intrusive mechanisms to avoid complex error handling routines and modification of production data. Solutions like lock could work but are limited to base libraries and wouldn't work in SPDE.

The solution we propose is non intrusive and automated. It uses the metadata definitions to identify which tables need to be reviewed, and then operating system commands to discover if the particular files are being locked or not. The next diagram illustrates how the information flows from the SAS metadata to the low level file locations for a particular schema.

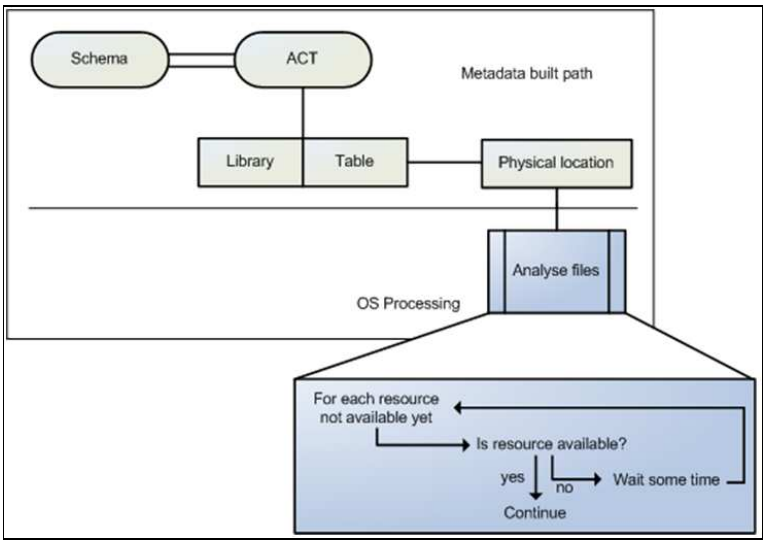


Figure 5: From metadata to physical files

1. Get the ACT associated to the schema we want to analyze. Similar to step 1 in Revoking access.

2. Get the list of objects that are under the selected ACT coverage. For this we will need to perform a query to the metadata server. Such query can be similar to this:

```
<GetMetadataObjects>
  <Reposid>repository_id</Reposid>
  <Type>AccessControlTemplate</Type>
  <Objects/>
  <NS>SAS</NS>
  <Flags>2309</Flags>
  <Options>
    <Templates>
      <PhysicalTable name="" id="" />
    </Templates>
    <XMLSelect search="AccessControlTemplate[@Name='MyACT']"/>
   </Options>
</GetMetadataObjects>
```

   This query returns the ACTs and associated tables. If we want to add other elements like cubes, we can add additional lines in the Templates tag:

```
<Cube name="" id="" />
```

3. For the tables, get the associated path. For that, we need first to know in which library the table is. This can be done with a query to the metadata server:

```
<GetMetadataObjects>
    <Reposid>&repository.</Reposid>
    <Type>SASLibrary </Type>
    <Objects/>
    <NS>SAS</NS>
    <Flags>2309</Flags>
    <Options>
    <Templates>
        <PhysicalTable name="" id=""/>
    </Templates>
    </Options>
</GetMetadataObjects>
```

This query will return a list of table-library which can be merged with the result from the previous step.
A sample of the output of this query would look like this:

	library	table	tableId	act
80	detail	bms_applicant_dim	A5HSO9KD.B900179N	BM Applications (04)
81	detail	bms_applicantaddress_dim	A5HSO9KD.B900177Q	BM Applications (04)
82	detail	bms_applicantrole_dim	A5HSO9KD.B900178O	BM Applications (04)
83	detail	bms_applicantverification_dim	A5HSO9KD.B900179D	BM Applications (04)

Now we can get the path of the detail library by querying the metadata or by having a lookup table with all the library-paths pairs. The second option is preferred here to keep the example simple. In our case we would have:

library	path
detail	/data/integrate/
mart	/data/exploit

Which means the absolute paths for the files will be:
detail.bms_applicant_dim: /data/integrate/bms_applicant_dim*
detail.bms_applicantaddress_dim: /data/integrate/bms_applicantaddress_dim*
and so on.

For cubes, we would need to do similar exercise and get the AssociatedFile component.

4. Once we have the paths, we can run an operating system dependant command to identify if the file is currently being used. In UNIX[3] environments we have the fuser command :

```
$ fuser -f -u /data/integrate/bms_application_dim*

/data/integrate/bms_application_dim.mdf.0.0.0.spds9:
1589382(userA) 2244752(userB)
```

In this case, two users (userA and userB) are using the table. The 1589382 and 2244752 numbers are the process ids.

We can capture the output of this command line with a pipe filename:

```
filename output pipe "fuser -f -u /test/data/integrate/file*";
```

Then we can read the output with an input statement to identify who is using that file.
In order to automate this step, we could incorporate the sleep function and repeat the process every n minutes until all resources has been released.

[3] For Windows environments you we could use NET FILE

3. WAIT UNTIL A RESOURCE IS FREE

This process would be similar to step 1 but removing the deny ACT and granting the original ACT back.

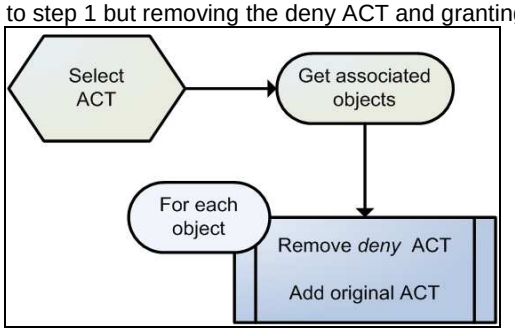


Figure 6: Granting permissions back

INTEGRATION WITH DI STUDIO JOBS

The previous steps have been encapsulated in two macros so that they can be easily integrated with the SAS Environment:

- `%setAccess(schema=SCHEMA_NAME, access=OPERATION);` This macro will restore the SCHEMA_NAME ACT to its current one (will remove the deny one, if it exists). There are two operations available: grant and remove. This macro would cover steps 1 and 3.
- `%checkUsage(schema=SCHEMA_NAME, retry=N, wait=T);` This macro will check if the SCHEMA_NAME associated files are being used. If one of the tables is locked, it will wait T minutes and retry. This operation will be repeated N times, after that time a message will be sent to the administrator and the process stopped.

ADMINISTRATION TOOL

An administration tool was provided to help in the maintenance of the ACTs. There are three main operations that can be done:

- List/View: this shows how the ACTs are distributed across the metadata.
- Add/Deny: allows the administrator to remove or grant access to one or many schemas.
- Management: allows the administrator to reset the permissions or revoke access to the whole environment.

The Add/Deny and Management tasks were performed using the above macros.

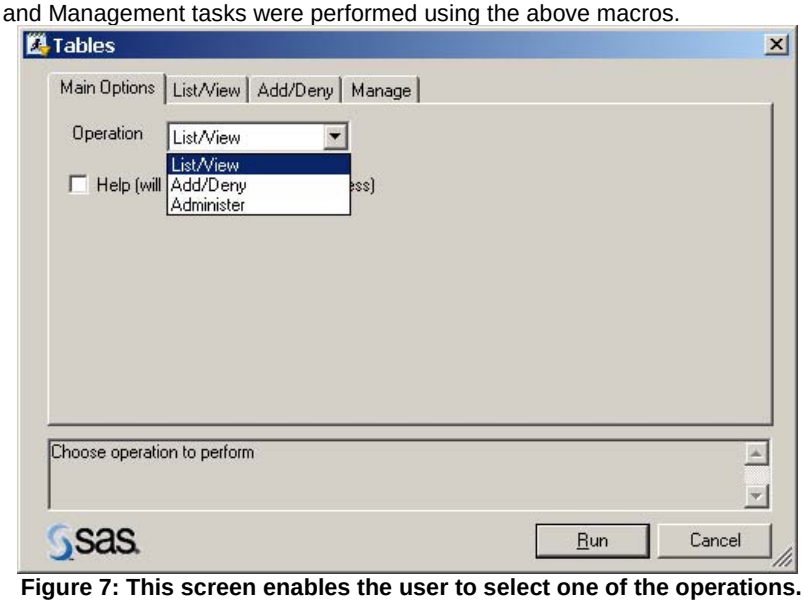
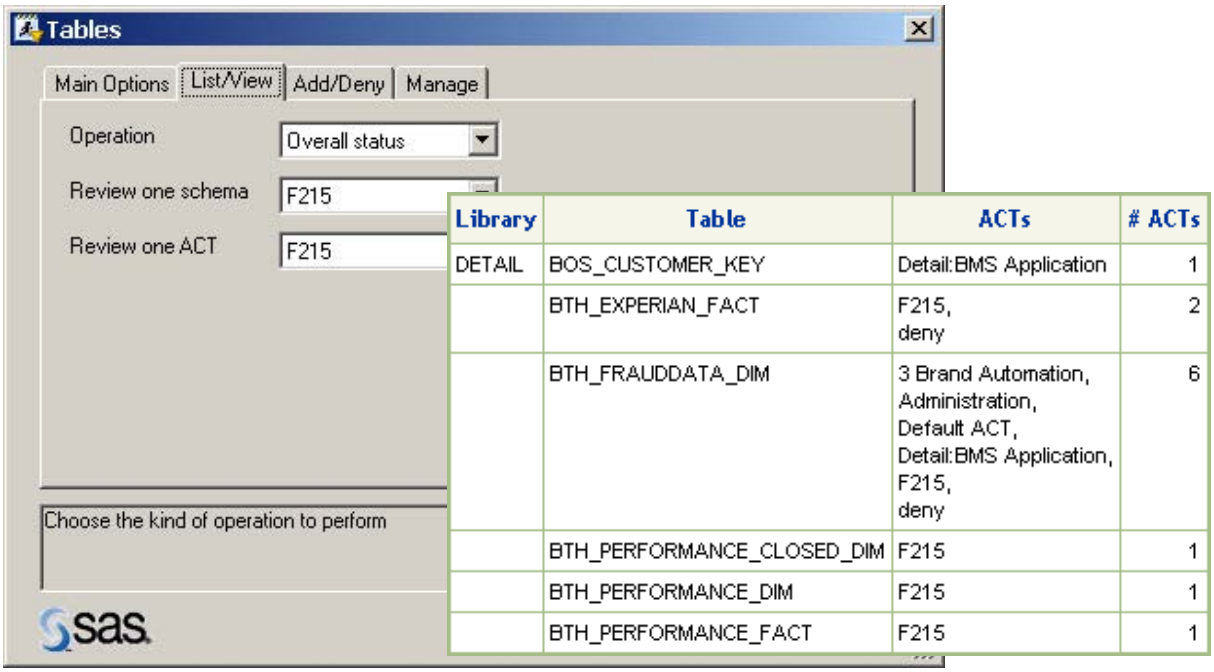


Figure 7: This screen enables the user to select one of the operations.



Library	Table	ACTs	# ACTs
DETAIL	BOS_CUSTOMER_KEY	Detail:BMS Application	1
	BTH_EXPERIAN_FACT	F215, deny	2
	BTH_FRAUDDATA_DIM	3 Brand Automation, Administration, Default ACT, Detail:BMS Application, F215, deny	6
	BTH_PERFORMANCE_CLOSED_DIM	F215	1
	BTH_PERFORMANCE_DIM	F215	1
	BTH_PERFORMANCE_FACT	F215	1

Figure 8: The list/view screen, with the output

In the above figure, we see several ACTs applied to tables. This is an example of how the metadata shouldn't be organized. The tool helps us identify inconsistent situations like for example the BTH_EXPERIAN_FACT that contains the F215 and the Deny ACT applied or the BTH_FRAUDDATA_DIM with six ACTs.

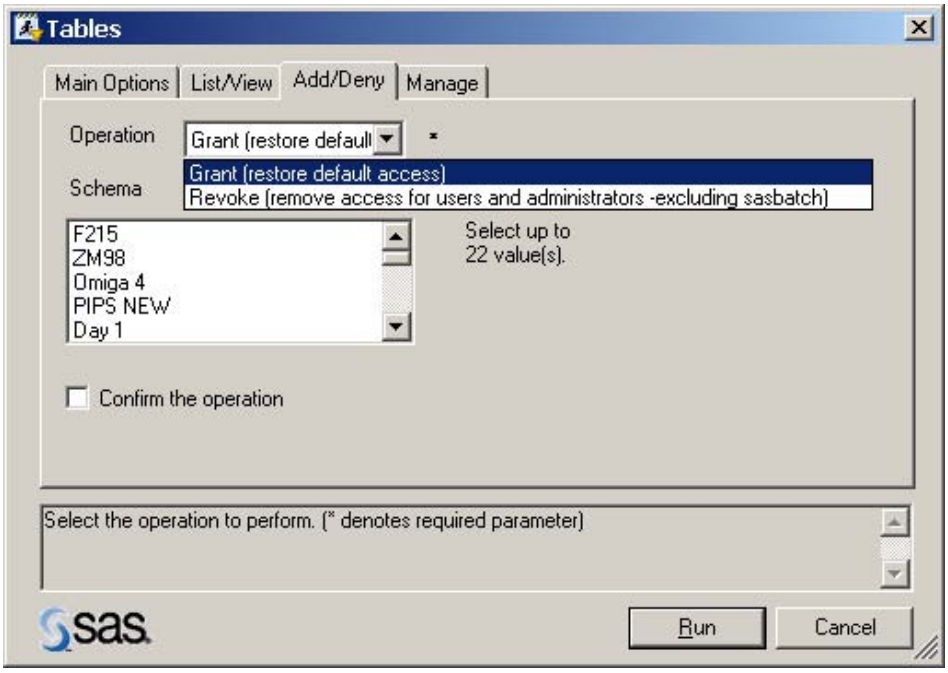


Figure 9: Add/deny screen

Two operations are available in the Add/Deny tab:

- Grant: If present, will remove the Deny ACT in the selected schema.
- Deny: Applies the Deny ACT to the selected schema.

This tab can help solve inconsistencies as shown in figure 8.

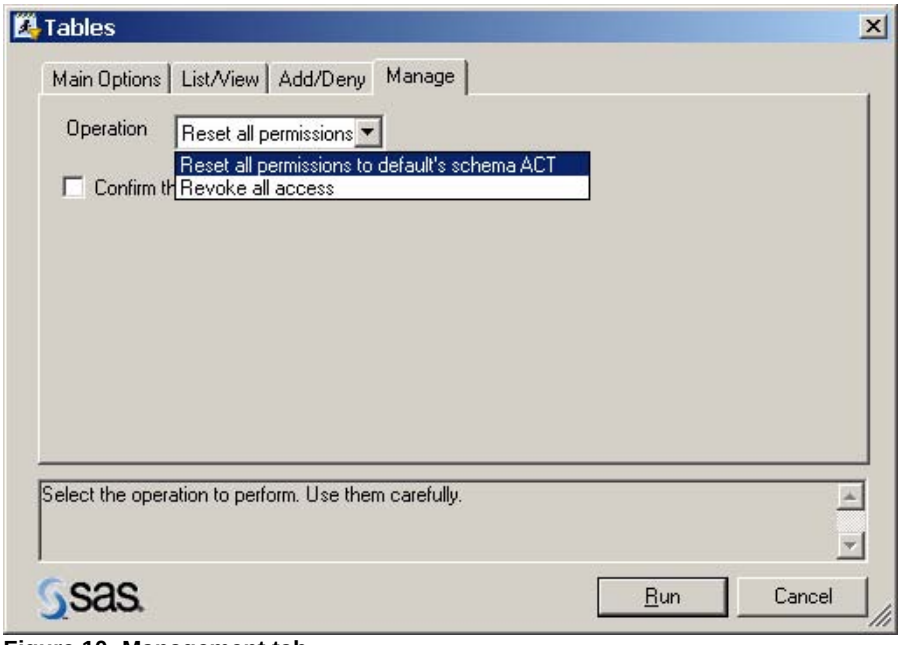


Figure 10: Management tab

The operations available in the Manage tab are:

- Reset all permissions to default's schema ACT: Goes through all objects for every schema and leaves the original ACT. This option would clean a situation like in the example in the previous page.
- Revoke all access: This is to shut down the environment forbidding access to all users.

IMPROVEMENTS AND CONCLUSIONS

A full automation was achieved but still a monitor on the application is needed in case the users don't release the tables. Several enhancements could be done to this model to tackle the limitations:

1. The step 2 waits until a resource is free. This could be problematic if the user doesn't release the table after some time (for example, he's in the middle of a query) and the administrator needs to contact him/her directly. A possible solution is to automatically send an email to the users that are locking the files so that they release them. After the schema has been updated we could send another email informing that the tables are updated and available again.

2. The amount of time that the full schema is not available depends on two factors: the updating time and the time that the users take until they release the resources. A solution for this could be to update the schema in temporary locations (like different libraries) and remove the access in the last part of the transaction just before the data gets copied to the final destination. In this case the amount of time that the resources are offline is minimized.

3. There is a clear dependency in the SAS Metadata Server. This means that if the server is temporarily paused or stopped (for maintenance or backup processing for example) this process will not work. A workaround can be made to identify if the metadata server is up and running using the METAOPERATE procedure to determine its availability and pause the process.

Automatic manipulation of metadata is something that can help us to have a better control of our environment. Although it can look a bit complex at the beginning, the benefits that can provide are worth spending some time in the learning curve.

REFERENCES

- SAS 9.1.3 XML Libname Engine: User's guide:
  http://support.sas.com/rnd/base/xmlengine/sxle913/usersguide913.htm

- SAS Open Metadata Interface
  http://support.sas.com/documentation/onlinedoc/91pdf/sasdoc_91/base_omi_7253.pdf

- SAS 9.1 Open Metadata Interface: User's Guide
  http://support.sas.com/documentation/onlinedoc/91pdf/sasdoc_91/base_omiug_7255.pdf

CONTACT INFORMATION

Any comments or questions are encouraged. Please contact the author at:

Jorge Jordá Morlán
Work email: JorgeMorlan@Hbosplc.com
Personal email: ingjjorda@yahoo.es
PO Box 81,
Pendeford Business Park,
Wobaston Road,
Wolverhampton, WV9 5HZ
UK

© 2009 Lloyds Banking Group plc and its subsidiaries.

SAS® and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

APPENDIX I: QUERYING SAS METADATA

Tasks in this paper involve metadata manipulation. This can be achieved by using XML based queries and PROC METADATA.
First we have open set up the connection to the SAS Metadata Server. To do that we will use the following code:

```
option metaserver=server_name
       metaport=metadata_port
       metauser=user
       metapass="password"
       metarepository="RepositoryName";
```

The next step is to query a XML string; you can use the following piece of code. This will create two temporary files and then copy the query code into the request (input) file. Then the proc metadata will execute the query and will return the result to the response file, in an XML format. You can choose to associate a physical file for request and response so that you can examine the XML input and output.

```
filename request temp;
filename response temp lrecl=65000;
data _null_;
       file request;
       put '<GetMetadataObjects>';
       put "    <Reposid>&repositoryId</Reposid>";
       put '    <Type>AccessControlTemplate  </Type>';
       put '    <Objects/>';
       put '    <NS>SAS</NS>';
       put '    <Flags>2309</Flags>';
       put '    <Options>';
       put '        <Templates>';
       put '               <AccessControlTemplate/>';
       put '               <PhysicalTable/>';
       put '        </Templates>';
       put '  </Options>';
       put '</GetMetadataObjects>';
run;
proc metadata in=request out=response; run;
```

The content of the response file needs to be transformed to a SAS dataset. For that we can use a SAS XML map (which will tell SAS how to read the XML file) and the xmlmap engine (which will use the map to actually read the file). You can use the SAS XML Mapper tool to build the reader but once you get confidence you can write it by hand.

```
data _null_;
file map lrecl=65000;
put '<SXLEMAP name="AUTO_GEN" version="1.2">';
put '  <TABLE name="listTables">';                                  (1)
put '  <TABLE-PATH syntax="XPath">                                  (2)
            /GetMetadataObjects/Objects/SASLibrary/Tables/PhysicalTable
       </TABLE-PATH>';
put '  <COLUMN name="libraryName" retain="yes">';
put '      <PATH syntax="XPath">
            /GetMetadataObjects/Objects/SASLibrary/@Name
       </PATH>';
put '      <TYPE>character</TYPE>';                                 (3)
put '        <DATATYPE>string</DATATYPE>';
put '        <LENGTH>32</LENGTH>';
put '  </COLUMN>';
put '  <COLUMN name="Id" retain="yes">';
put '      <PATH syntax="XPath">
            /GetMetadataObjects/Objects/SASLibrary/Tables/PhysicalTable/@Name
       </PATH>';
put '      <TYPE>character</TYPE>';
put '      <DATATYPE>string</DATATYPE>';
put '      <LENGTH>17</LENGTH>';
```

```
        put '  </COLUMN>';
        put '  </TABLE>';
        put '</SXLEMAP>';
run;
libname response xml xmlmap=map access=READONLY;
```

Three important parts need discussion in this code:

1. This defines the output table name. Needs to meet SAS table naming convention.
2. The path attribute defines the observation boundary for the dataset. In this example, the engine will produce one observation for each PhysicalTable.
3. For each required column we need a COLUMN element which will contain the variable definition like the name, the retain attribute, the XML path, etc.

Finally, the libname statement will assign a virtual library that will contain the listTables dataset.