This document describes UNLV’s technical approach to the development of an extensible, institutional Metadata Repository. The repository supports the development of multiple subject areas independently, while maintaining the ability to relate any item to any other, including the tracking of lineage relationships.

Mike Ellison, Christina Drum
UNLV Office of Institutional Analysis and Planning
5/11/2012
UNLV – Metadata Repository Design Concepts

As a central responsibility in its ongoing efforts to develop an institutional data warehouse and business intelligence initiative, UNLV maintains and develops an institutional Metadata Repository. Subject areas of immediate interest include Data Definitions (elements identified as having strategic value to the institution), Relational Database Metadata (RDBMS systems, tables, and columns), ETL metadata (data warehousing jobs and sequences), and Reporting Elements metadata (OBIEE presentation elements). Additional areas of interest to be developed include Business Process metadata, the tracking of Data Feeds, and the application of Stewardship.

UNLV maintains all subject areas of metadata in a central SQL Server database, designed around a type/subtype paradigm borrowed from object-oriented programming. Several key considerations influenced the approach of this design:

1. Ultimately we want multiple subject areas of metadata.
2. Time/resource constraints require us to build this out over time, so we need an approach that allows for expansion.
3. Each subject area of metadata should stand alone, potentially with its own set of applications.
4. We do however want the ability to establish associations among any of the different subject areas.
5. We want the ability to track a complete lineage of relationships across subject areas.
6. Some subject areas may be refreshed or updated via purely automated processes. Some must be manually tracked. We want the ability to work with both.

Type/Subtype Paradigm

To allow for multiple subject areas of metadata to be built independently, but still remain inter-relatable, the metadata repository is designed around a type/subtype object concept. One table, BaseObjects, serves as a master catalog for all metadata records and contains a few attributes common to all types of metadata:

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BaseID</td>
<td>Auto-incrementing identity column.</td>
</tr>
<tr>
<td>ObjectType</td>
<td>Up to 8-character text code which identifies the subject area for the given object record</td>
</tr>
<tr>
<td>ObjectKey</td>
<td>A key unique within a given subject area; the combination of ObjectType and ObjectKey is unique across the table.</td>
</tr>
<tr>
<td>Name</td>
<td>The title of the object as it appears in the catalog.</td>
</tr>
<tr>
<td>Description</td>
<td>A friendly text description of the object.</td>
</tr>
</tbody>
</table>

The BaseObjects table serves two purposes. In addition to functioning as the master catalog for all metadata objects, it also, through the BaseID field, provides the means for which global ID values are assigned to individual items of metadata. These global ID’s are intended to be unique across all subject
areas. In our SQL Server implementation, we use an identity column to auto-increment the values as new items are added to the table. A sequence could be used in an Oracle implementation.

For any given subject area of metadata, a second table is used to hold the specific attributes associated with items of that metadata type. For example, to store information about Data Definitions, we create a subtype table called DataDefinitions_Tbl. The “Tbl” suffix is a convention used throughout the repository to denote a table that relates to BaseObjects but holds type-specific attributes. While Name and Description are maintained in the central BaseObjects table, attributes specific to Data Definitions, such as Interpretation, PotentialValues, and SourceDescription are maintained in the DataDefinitions_Tbl subtype table. The two are then related through the common BaseID field as the primary key in each.

To ensure that as records are added or updated the data attributes are appropriately divided between the two tables, all data entry, manipulation, and selection is handled through a dedicated view. By convention, the view name is the same as the subtype table without the _Tbl suffix. For example, there is a view titled DataDefinitions which provides the primary interface for selecting and manipulating records in both the BaseObjects and DataDefinitions_Tbl tables. The tables themselves are intentionally off-limits for direct editing. Instead we use triggers coded on the view to support editing in such a way as to put the right data in the base and subtype tables.

Example – Tracking Widgets
Say we want to track an area of metadata called widgets. (Bad name, I know. Pretend it is a very important metadata subject area of interest.) We start by creating the BaseObjects table and a couple of related tables for tracking object types and categories. INSERT statements in the example prepare an object type code ‘TESTWIDG’ to identify the new metadata area.

```sql
create table ObjectTypeCategories
(
    CategoryCode varchar(8) primary key
,    Category varchar(64)
,    CategoryDesc varchar(1024)
)
GO

create table ObjectTypes
(
    TypeCode varchar(8) primary key
,    CategoryCode varchar(8) foreign key references ObjectTypeCategories(CategoryCode)
,    TypeDesc varchar(128)
,    TypeDescShort varchar(32)
,    DetailsTable varchar(128)
,    DetailsView varchar(128)
)
GO

-- test data
insert into ObjectTypeCategories values ('TEST', 'Test Category', 'This category is for test objects.')
go

insert into ObjectTypes
    values ('TESTWIDG', 'TEST', 'Test Widget', 'TestWidgets_Tbl', 'TestWidgets')

create table BaseObjects
```
We also create a table to hold the attributes that would be specific to a widget. Note the unique key applied to the WidgetID field. It is important that there be a field that can serve as a unique subject area key. While BaseID will be the globally unique key across all subject areas, WidgetID will be the unique key within the subject area.

```sql
-- create the table to store detail information about TestWidgets.
-- By convention, end this in _Tbl
create table TestWidgets_Tbl
(
    BaseID int primary key foreign key references BaseObjects(BaseID) on delete cascade
, WidgetID int
, ManufactureDate datetime
, Manufacturer varchar(32)

, CONSTRAINT cnst_testWidget_WidgetID UNIQUE (WidgetID)
)
GO
```

Finally, we create the view intended to be the primary interface for adding, modifying, and deleting widgets. The view selects columns from both the BaseObjects and TestWidgets_Tbl tables, ensuring that the full set of editable attributes for the widget is available in one database object.

```sql
-- create the view to display detail TestWidget data with corresponding base data. By convention, we won't include a suffix (we want to treat it as though it is a table).
create view TestWidgets
as
select d.BaseID
, b.Name, b.Description
, d.WidgetID, d.ManufactureDate, d.Manufacturer
from TestWidgets_Tbl d inner join BaseObjects b on d.BaseID = b.BaseID
GO
```
To make the metadata view fully interfaceable we code INSTEAD OF INSERT, UPDATE, and DELETE triggers on the view. The INSTEAD OF INSERT trigger adds a record for the metadata item to the BaseObjects table, generating the new primary key BaseID for the item. It then in turn uses this newly generated key to insert a record in the subtype TestWidgets_Tbl table. The INSTEAD OF UPDATE and INSTEAD OF DELETE triggers similarly work to affect both tables appropriately. Note the use of the ObjectKey field – the field in BaseObjects which is intended to be unique within a given ObjectType - in each trigger. This key mechanism matched to the subject area key WidgetID is used to keep records synchronized across both tables.

```
CREATE TRIGGER TestWidgets_Trigger_Insert_InsteadOf
ON TestWidgets
INSTEAD OF INSERT AS
BEGIN

declare @typeCode varchar(8)
set @typeCode = 'TESTWIDG'
-- insert the base record(s) first so we can generate BaseID's;
Insert Into BaseObjects (ObjectType, ObjectKey, Name, Description)
Select @typeCode , convert(varchar(64),WidgetID) , Name , Description
from inserted

-- now link the BaseObjects data back to [inserted] to join the new BaseID
-- and insert the rest of the details in TestObjects_Tbl
Insert Into TestWidgets_Tbl (BaseID, WidgetID, ManufactureDate, Manufacturer)
Select b.BaseID , i.WidgetID, i.ManufactureDate, i.Manufacturer
from inserted i inner join BaseObjects b on convert(varchar(64),WidgetID) = b.ObjectKey
where b.ObjectType = @typeCode

END
GO
```

```
CREATE TRIGGER TestWidgets_Trigger_Update_InsteadOf
ON TestWidgets
INSTEAD OF UPDATE AS
BEGIN

-- trying to update BaseID?  Don’t allow that
if (UPDATE(BaseID))
begin
raiserror(N'You can not edit the BaseID.', 16, 1)
return
end

-- other base fields are okay
if (UPDATE(Name)) begin
Update BaseObjects Set Name = i.Name
from inserted i inner join BaseObjects b on i.BaseID = b.BaseID
where i.BaseID = b.BaseID
end
```

if (UPDATE(Description)) begin
  Update BaseObjects Set Description = i.Description
    from inserted i inner join BaseObjects b on i.BaseID = b.BaseID
    where i.BaseID = b.BaseID
end

-- now look to the detail fields
-- by convention, if the WidgetID changes, update the ObjectKey in the base to keep them in sync
if (UPDATE(WidgetID)) begin
  Update TestWidgets_Tbl Set WidgetID = i.WidgetID
    from inserted i inner join TestWidgets_Tbl b on i.BaseID = b.BaseID
    where i.BaseID = b.BaseID
  Update BaseObjects Set ObjectKey = convert(varchar(128), WidgetID)
    from inserted i inner join BaseObjects b on i.BaseID = b.BaseID
    where i.BaseID = b.BaseID
end

-- and address the other detail fields
if (UPDATE(ManufactureDate)) begin
  Update TestWidgets_Tbl Set ManufactureDate = i.ManufactureDate
    from inserted i inner join TestWidgets_Tbl b on i.BaseID = b.BaseID
    where i.BaseID = b.BaseID
end

if (UPDATE(Manufacturer)) begin
  Update TestWidgets_Tbl Set Manufacturer = i.Manufacturer
    from inserted i inner join TestWidgets_Tbl b on i.BaseID = b.BaseID
    where i.BaseID = b.BaseID
end

END
GO

-- create a DELETE INSTEAD OF trigger on the view
CREATE TRIGGER TestWidgets_Trigger_Delete_InsteadOf
ON TestWidgets
INSTEAD OF DELETE AS
BEGIN
  -- actually delete from the BaseObjects table.
  -- cascade delete will take care of removing other referenced records
  delete from BaseObjects
    where BaseID in (select BaseID from deleted)
END
GO

There may be significant work to establish the subtype table, view, and triggers, but once that work is complete, the following benefits are gained:

1. All items of metadata are centrally cataloged in the BaseObjects table and given a BaseID value that will be unique across all metadata subject areas. This is important to build inter-relationships among different subject areas and establish flexible lineage relationships.
2. The TestWidgets view may now be treated exactly like a table. One may use plain SQL to insert, update, or delete from it, and the BaseObjects data remain in sync.
3. The TestWidgets view may also be exposed to any application development context. One could create an application that manages TestWidgets data independently of other metadata. One could also use an ODBC connection to TestWidgets for editing or reporting contexts.

4. The TestWidget coding serves as a template for other subject areas. After building one, the others are easier.

It is also useful to know that, for the TestWidget subject area, only the main TestWidgets table (view) requires this level of attention as the only object data that will be maintained in the master catalog. Other tables that would provide related data to TestWidgets may be built as normal tables and related to TestWidgets on the subject area key WidgetID. For example, consider the following schema that links a widget category and a widget supervisor to the main TestWidget table:

```
TestWidgets
| BaseID | Name | Description | WidgetID | ManufactureDate | Manufacturer |
TestWidgets_Category
| WidgetID | Category |
TestWidgets_Supervisor
| WidgetID | SupervisorID | SupervisionType |
Supervisors
| SupervisorID | LastName | FirstName | Phone | Email |
```

All tables within the subject area relate on the subject area key WidgetID. The only table with the BaseID identifier, and a record in the main catalog BaseObjects, is the main TestWidgets table itself. In this way, a complete metadata subject area may be easily developed independently of others, while preserving its place in the master catalog.

**Metadata that can be Automatically Refreshed**

The following are metadata areas that can be refreshed (kept up-to-date) through at least somewhat automated strategies:

<table>
<thead>
<tr>
<th>Metadata Area</th>
<th>General Strategy for Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relational Databases, Tables, and</td>
<td>Execute queries against INFORMATION_SCHEMA views to retrieve</td>
</tr>
<tr>
<td>Columns</td>
<td>database, table, and column metadata</td>
</tr>
<tr>
<td>PeopleSoft Metadata</td>
<td>Query PS metadata tables such as PSRECDEFN, PSDBFIELD, PSPNLFIELD to retrieve record, field, and page information.</td>
</tr>
<tr>
<td>DataStage ETLs</td>
<td>Use the Export XML utility in the DataStage Manager application to generate xml representations of desired ETL’s; write scripts to parse the resulting XML to extract ETL metadata, including source and target database tables</td>
</tr>
<tr>
<td>-----------------------------------------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>OBIEE Repository Layers (Physical, Logical, Presentation)</td>
<td>Use the Generate Metadata Dictionary utility in the OBIEE Administrator application to generate xml representations of desired repository layers; write scripts to parse the resulting XML to extract OBIEE metadata, including relationships of objects among layers.</td>
</tr>
</tbody>
</table>

For any metadata that can be automatically refreshed, we build staging tables in the metadata repository to correspond to the type/subtype metadata views holding the active data. These staging tables are periodically truncated and loaded with up-to-date metadata through whatever automated strategy makes sense. A stored procedure is then run to compare the existing active metadata to the newly staged set based on the subject area key. The stored procedure updates or adds any changed metadata records (without disturbing the BaseID global key in the case of updates).

**Metadata that is Manually Tracked**

Data definitions, business processes, data feeds, and stewardship are examples of metadata which must be manually cataloged and updated over time. Rather than using staging tables, we build applications to manage the live data for these subject areas. Applications may be specifically targeted to a given subject area, and may be used to manage relationships among subject areas.

**Designing for Associations among Metadata Areas**

Regardless of whether metadata is automatically refreshed or manually tracked, the application of the globally unique BaseID allows any item to relate to any other.

Consider the example of a Data Definition, which records a strategically-valuable informational element and instructions on how to derive the element from transactional sources. Though the Definition is conceptual, it is ultimately implemented in the data warehouse, with logic in an ETL writing a derived result to its own database column. It is desirable to relate the Data Definition item with the ETL and the Database Column items.

As we determine specific areas where such associations make sense, we build those in as their own detail tables in a subject area schema. For example: *DataDefinitions* (the metadata view for DataDefinitions_Tbl) has a related table *DataDefinitions_Implementations*, with two columns: DefinitionID and ImplementationID. Each column is a foreign key to the BaseID column in the BaseObjects table. DefinitionID holds the BaseID for a specific Data Definition record, and ImplementationID holds the BaseID value for a related Database Column, OBIEE Presentation Element, or ETL job. In this way we can track implementations related to a given Data Definition, regardless of the implementation’s type.
Stewardship provides another metadata area which may be associated to any of a number of other metadata items. An Operational Steward may be associated to a Business Process or a Data Definition. A Technical Steward may be associated to a Data Feed, or a Data System. Tables holding information about individuals and their roles in the institution may be maintained in the central Stewardship subject area, while specific associations to other metadata items may then be recorded in a Stewardship_Responsibilities table, using the BaseID of objects to make the relationship.

**Lineage**

Lineage is a series of associations across different subject areas, with each association in the chain implying a specific atomic relationship between two items of metadata. The goal is to track how a given metadata item (like a database table) may be a source to another (like an ETL job) which itself may be a source to another (like a target database table) and so on (the target table is a source to a reporting application, etc.) We define lineage as a series of ancestor-descendant relationships, with one item clearly defined as the ancestor (source to) and the other its descendant (target of) at any stage in a lineage chain.

Our data lineage flows in this general format:

- PeopleSoft Pages host PeopleSoft Records (Pages are ancestors, Records are descendants)
- PeopleSoft Records are instantiated as Database Tables (Records are sources - ancestors - for Tables, their descendants)
- Database Tables are sources for ETL jobs (Tables are ancestors to ETL jobs)
- ETL jobs then write to different Database Tables (ETL jobs are ancestors to additional Tables)
- Those Database Tables are then sources to an OBIEE repository Physical Layer (Table is the ancestor; a Physical Layer is the descendant). Those Database Tables may also be sources to additional ETL jobs that provide additional data processing and restructuring.
- An OBIEE Physical Layer is a source to a Logical (Business Model) Layer (Physical is the ancestor, Logical is the descendant)
- That Logical Layer is a source to a Presentation Layer (Logical is the ancestor to its descendant Presentation)
- That OBIEE Presentation Layer may then be a source to specific institutional reports or dashboards (the Presentation Layer is an ancestor, with specific Reports/Dashboards as descendants)

We maintain these relationships in a single table called Lineage. This table has two columns, simply Ancestor and Descendant, each a foreign key to the BaseID column of the BaseObjects table. This gives the most flexibility to allow any metadata item desired to participate in a lineage chain. The following table shows an example of how these lineage relationships are represented in the Lineage table:
As an example, take item 594, the descendant in the last row and trace its ancestry up the table. Its immediate parent is item 415, which itself is derived from items 305 and 312. Item 312 has no additional ancestors, but item 305 does with its immediate parent 207. Item 207 is then a child of item 101. The full lineage from item 101 to item 594 then looks like this:

101 → 207 → 305 → 415 → 594.

Item 101 may be a PeopleSoft Record, 207 a Database Table, 305 an ETL job, 415 a target Database Table, and 594 a reporting application leveraging that table. This example illustrates two very powerful benefits of maintaining lineage:

- We can trace back any reporting element not just to its source database tables, but beyond that right to the PeopleSoft editing interfaces. This provides a quick way to identify the sources for reporting elements.
- We can trace items forward in the lineage as well. If, for example, modifications were made to a given PeopleSoft page or record, it would be a simple query to determine what Database Tables, ETL jobs, and Reporting elements could be affected by the modifications.

To achieve a hierarchical result set that spans the full lineage tree, we query the Lineage table using recursive *common table expressions* in SQL. The following SQL shows a query to display all ancestors for a given metadata item:

```sql
with Ancestors as
(
    -- anchor
    select r.Ancestor, r.Descendant, -1 as Level, r.Ancestor as BaseID,
         b.Name, b.Description, b.ObjectType, b.ObjectKey
    from Lineage r inner join BaseObjects b on r.Ancestor = b.BaseID
    where r.Descendant = @baseID

    -- recurse
    union all
    select r.Ancestor, r.Descendant, Level - 1 as Level, r.Ancestor as BaseID,
         b.Name, b.Description, b.ObjectType, b.ObjectKey
    from Lineage r inner join BaseObjects b on r.Ancestor = b.BaseID
    inner join Ancestors a on r.Descendant = a.Ancestor
)
```
Likewise, we use the following to query for an item’s descendants:

```sql
-- @baseID is a parameter established elsewhere
-- which represents the metadata item for which we wish to
-- see its descendants.

with Descendants as
(
    -- anchor
    select r.Ancestor, r.Descendant, 1 as Level, r.Descendant as BaseID
    , b.Name, b.Description, b.ObjectType, b.ObjectKey
    from Lineage r inner join BaseObjects b on r.Descendant = b.BaseID
    where Ancestor = @baseID

    -- recurse
    union all
    select r.Ancestor, r.Descendant, Level + 1, r.Descendant as BaseID
    , b.Name, b.Description, b.ObjectType, b.ObjectKey
    from Lineage r inner join BaseObjects b on r.Descendant = b.BaseID
    inner join Descendants a on r.Ancestor = a.Descendant
)

select distinct Level, BaseID, Name, Description, ObjectType, ObjectKey
from Descendants
order by Level
;```

Because `BaseID` values are global across the entire metadata repository, any metadata item may serve in an ancestor-descendant relationship with any other. This is very flexible, but does introduce some considerations:

- If not careful, it is possible to introduce circular relationships in the `Lineage` table. Having the same item recorded as both the ancestor and descendent in the same lineage record is an obvious example. A less obvious, but more likely example could be an ETL job that sources from and writes to the same table. Or this example: table A is a source to an ETL which writes to table B... table B is the source to another ETL which writes to table A. It is important to ensure that processes which populate the `Lineage` table account for these potential circular relationships.

- It is possible to record lineage relationships that skip intervening levels. Borrowing from the above example, one could write a lineage row with PeopleSoft Record 101 as the ancestor and Report 594 as the descendent. If the intervening levels of metadata have not been developed out, this might even be desirable -- a short-term solution until the rest of the metadata is fleshed out. But as additional metadata areas come online, it is preferable that the ancestor-descendant records reflect immediate, atomic relationships to the extent possible.
To help address these concerns, we populate our Lineage table exclusively through a stored procedure. This stored procedure truncates the table and repopulates it according to current metadata entries and relationships, with logic that imposes rules designed to prevent circular references. We intend to update this procedure as new metadata areas are developed, to position them correctly in the full lineage tree.

**Summary**

UNLV’s approach to the design of its Metadata Repository reflects the desire to build out additional subject areas over time, while allowing for associations among items in different metadata areas. Attributes that are common to all metadata items, such as Name and Description are maintained in a central BaseObjects table. This table also maintains the BaseID field, a primary key that is globally unique across all metadata areas. The BaseObjects table also serves as a master catalog of all metadata items in the system.

For each metadata type, a second subtype table holds the attributes appropriate for it, and is named with a _Tbl prefix by convention. A metadata view then combines the base attributes with the subtype attributes into a single object, with triggers coded to make it fully interfaceable. Subject areas may be developed independently and make use of a subject-area key for relating data within the metadata area. The BaseID global key may then be used to create relationships across subject areas. This is particularly useful for establishing lineage relationships with as much flexibility as possible.