Schema Driven Assignment and Implementation of Life Science Identifiers (LSIDs)

Daniel Miranker, Sapna Bafna and Julian Humphries*
Department of Computer Sciences, University of Texas at Austin
*Department of Geological Sciences, Jackson School of Geosciences
{miranker, sapna}@cs.utexas.edu

Abstract

Life sciences identifier (LSIDs) is a formal global unique identifier standard intended to help rationalize the unique archival requirements of biological data. We describe an LSID implementation architecture such that data managed by a relational database management system may be integrated with the LSID protocol as an add-on layer. The approach requires a database administrator (DBA) to specify an *export schema* detailing the structure of the archived data, and a mapping of the existing database to that schema. This specification is accomplished through the equivalent of SQL view definitions. In effect, we define a domain specific language for implementing LSIDs. We describe the mapping of the view definition to an implementation as a set of database triggers and a fixed runtime library. This suggests a compiler for the domain specific language could be written that would reduce the implementation of LSIDs to the task of writing SQL view definitions.

Key words: LSID, metadata, RDF, resolution, trigger, view

1 Introduction

The management of biological data is subject to a centuries old social contract [7]. Reported results of experimental accomplishment must be backed up with sufficient information to duplicate the results. If other scientists are not able to duplicate a result, the published scientist may be called upon to provide further details. Failing that, the scientist is open to accusations of scientific misconduct or worse. Prior to the genomic revolution, this contract was fulfilled exclusively by maintaining a laboratory notebook where the design and preconditions of an experiment were scribed as were the measurements and observations of the experiment itself. Even today, the grading of undergraduate students' laboratory notebooks is central to the training of experimental scientists and may take on a ceremonial quality.

Modern equipment in research laboratories produce data at a rate such that data in computer files are supplanting laboratory notebooks. The data may be structured and managed by a database management system (DBMS). Independent of structure and management, the data is often available through a web site for review or additional analysis. This practice, despite its popularity, does not fulfill the social contract, URLs become stale, and web sites regularly vanish.

In May 2004, as a means to provide a contemporary method to support the social contract the Object Management Group (OMG) adopted the V1.0 specification of a global unique identifier

protocol called Life Science Identifiers, (LSIDs) [1]**.** The protocol specifies web services for authorities to establish directories of providers supporting the *assignment* of LSIDs, their later *resolution* to data and mapping individual LSIDs to service providers. The protocol stipulates that once data is associated with an LSID that data is immutable and will be forever available. Data may be updated through versioning, provided the integrity of old versions is also maintained, forever.

Our perspective is that LSIDs are surrogates that can be used to reference data independent of time and place, reminiscent of the use of row-ids and foreign keys as surrogates in a database. The differences being that data is write-once and Internet protocols enable the replication and migration of the database servers. The LSID protocol specification is structured such that individual laboratories may make a commitment to maintaining the longevity and integrity of their own data or data centers may accept third party data and maintain the commitments. Since funding and tenure of an individual laboratory is undependable there is some expectation in the Bioinformatics community that organizations such as the United States National Center for Biological Information (NCBI) and/or NSF supercomputer centers will emerge as general-purpose archival repositories of biological data referencable by LSID.

Our perspective is consistent with IBM's LSID best practices document [2]. Where we differ is in our goals. The intent of IBM's best practices document is to illuminate and facilitate the implementation of the protocol by engineers (and we are indebted to its authors).

A goal of our system is to facilitate the adoption of LSIDs by enabling the administrators of the databases that are proliferating in biology laboratories to easily add LSIDs to their publicly available data, and also make it convenient to move that data and their identifiers to permanent third-party repositories. By *easy*, we mean in an afternoon, and recognizing that these database administrators are typically graduate students in Biology whose knowledge of databases is self-taught.

To achieve these goals we define a domain-specific language that can be used to specify an export schema. The schema defines the records, simple or complex, that need to be assigned LSIDs. Our system only needs the base database schema and export schema to internally generate a layer on top of the existing database, which assigns LSIDs and persists data. Thus, a DBA only needs to specify an appropriate export schema to assign LSIDs to existing and new data.

The following section gives an overview of the LSID protocol while Section 3 illustrates some use cases of LSIDs in the bioinformatics world. Section 4 details the architecture and implementation of our system for implementing LSIDs. Some LSID resolution and metadata issues are discussed in Section 5 while Section 6 gives the status our implementation. Finally, we conclude in Section 7 with some pseudo code in the Appendix.

2 Overview of the Protocol

The LSID object model specifies *assignment, discovery,* and *resolution* services and their web service bindings for each of, SOAP, http GET and FTP [1]. *Assignment services* are used to synthesize an LSID identifier string and associate it with a data set. The assignment services are

structured to enable both stand-alone implementations of the protocol or centralized third party implementations. For the latter a data provider submits a data record to the assignment service and the service returns an LSID string.

Discovery services are UDDI like authorities that map LSIDs to resolution services. A single LSID may be supported by more than one resolution service. These services may differ by location and/or syntactic interface. Irrespective of the location or syntax, the data must be identical and immutable. Since the protocol stipulates that data is not updated in place, but by creating new versions, maintaining consistency of replicated resolution services is not difficult. In an effort to bootstrap the process, domain names are included in the LSID string and until UDDI like discovery services are established implementations simply use domain name mapping and SRV records as the discovery service [2]. This has had the unfortunate consequence of making LSIDs look like URLs rather than an arbitrary GUID string, creating confusion with respect to an LSIDs content and durability. See Figure 1.

Resolution services dereference an LSID, yielding the archived data. The two primary resolution service methods are getData (LSID) and getMetaData (LSID). The specification does not specify the content or syntactic structure of either the data of the metadata. Generically the specification anticipates that the MetaData will detail a data schema for the data expressed as RDF, XMI or other modeling language including XML schemas. An assignment/resolution service structured this way results in an LSID encapsulating an instance of a data set and the syntax and semantics necessary to integrate that data into a larger computation. A single resolution service may provide the data and meta-data in multiple syntaxes.

Format: urn:lsid:<domainName>:<namespace>:<objectId>[:<revisionId>]
Example: urn:lsid:ncbi.nlm.nih.gov:pubmed:12571434 //referencing a PubMed article

Figure 1: Format and Example of an LSID [1]

"Persistence: It is intended that the lifetime of an LSID be permanent. That is, the LSID … may be used as a reference to an object well beyond the lifetime of the object it identifies or of any naming authority [1].

It follows from the protocol that once assigned an LSID, a datum is immutable and persists forever. Sequential versioning of data is supported. Each version is similarly immutable and persistent. An updated datum results in an incremented version number. If data for an LSID are requested without a version number, then the most recent data version is returned. A version number may be specified, forever, and that version of the data will be returned.

3 Use Cases

3.1 Illustrative Use Case

Our application context is the NSF's "Assembling the Tree of Life " (ATOL) grand challenge. The grand challenge faced is in describing 5 to 10 million extant species, and computing and analyzing a unified phylogenetic tree. The effort spans organisms as far ranging as bacteria, plants and mammals.

Numerous projects are currently building labeled image databases, each geared to specimens in their specialty e.g. mycology, the study of fungi. Similarly, they are organizing the terminology of their corpuses as ontologies and working to exploit Internet technology to tie this information together and make it highly available. In addition to these efforts, work is underway to build web services for the computation of trees and the archival storage and retrieval of completed studies. In short, this science, Systematics, is moving toward a distributed scientific workflow.

The illustrative use case concerns rendering the annotations that label the edges of computed phylogenetic trees. A phylogenetic tree is, among other things, a hierarchical clustering of a data set. Each element of the set is a feature vector describing a specimen. Each dimension of the feature vector is called a *character*, and the value of the character for a specimen a *character state*. The ensemble is known as a *data matrix*. The set of specimens define the rows, characters define the columns and the matrix cells are filled with *character states*. In a study each specimen is chosen as an exemplar of a taxon, or in lay terms, species. In informal discussions a set of rows may be referred to as a set of taxa.

Character states may be the labels for individual residues in gene or protein sequences (e.g. A,C,G,T) or they may be integer coded descriptions of morphological properties e.g. the shape of a tooth, or the range of motion of a joint. While gene sequences are proving most effective at resolving the structure of a phylogenetic tree, much of the biological interpretation still rests on the richer description of the specimens. Ultimately, the connection of molecular phenomenon with the evolutionary development of phenotype promises to be a watershed of information. For example, a study may reveal differences in metabolic pathways that distinguish cacti from tropical plants and suggest ways of genetically engineering drought resistant food crops.

A present difficulty in the analysis of a computed tree is that there is no operable linkage between the trees and the wealth of information concerning the taxa and their state definitions. When visualizing a tree this problem manifests as integers and letters serving as edge labels. This is important as those labels represent the distinguishing character states separating one group of organisms from one another, at least one for each vertex in the tree. The scientific importance of each *split* can only be, (and must be), evaluated with respect to the definition of the character states, e.g. plants with wide leaves vs. narrow leaves. In the current infrastructure, an evaluation can only be made by accessing source material. Although systematic biologists no longer have to go to walk to a library to locate the supporting information, that is their only progress into distributed Internet computing.

LSIDs are emerging as the method of choice for creating the operable linkages across this workflow. Since an LSID is a string of fixed length, extending regularly structured coded information, (i.e. integer coded matrices) retains a regular structure, facilitating the upgrade of legacy software.

3.2 UT CT LSID Prototype

The University of Texas UTCT Data Archive (UTCT) [8] is an advanced prototype of the well-known Digimorph [9] repository. It stores metadata about specimens and both metadata and actual slice images from high-resolution X-ray computed tomographic scans of those specimens.

Image data are stored in an SQL server database and currently comprise over 130k images. Since UTCT serves as a good archival database of biological specimens we are experimenting with our LSID prototype utilizing this system.

For our LSID prototype we furnish a hierarchy of archived data. We export each specimen as its Darwin Core record [12]. Darwin Core is a simple set of data element definitions designed to support the sharing and integration of primary biodiversity data. It is a standard supported by the Global Biodiversity Information Facility (GBIF), an organization sponsored by an international treaty concerning biodiversity. A goal of Darwin Core is to support the integration of the world's biological collections' on-line biodiversity catalogs.

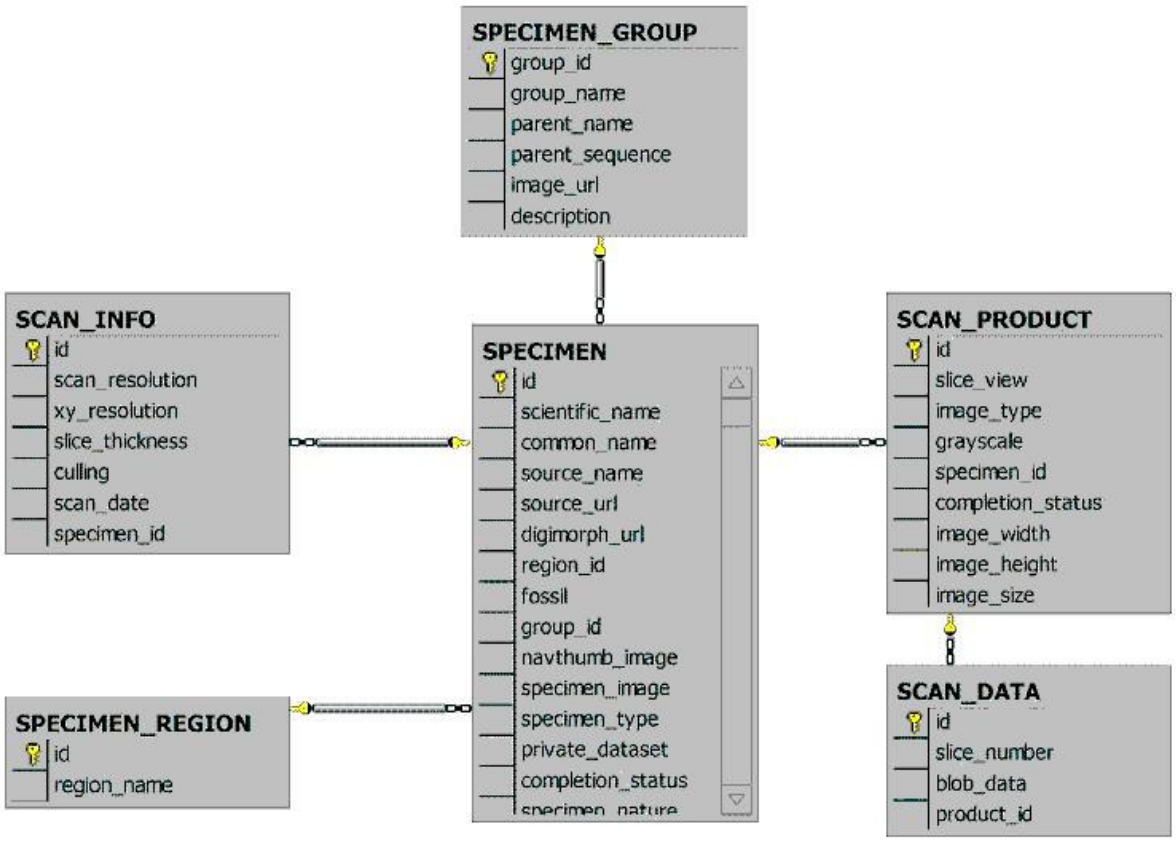

Figure 2: Data model of the relevant portions of the UT CT database schema, we wish to embellish with LSIDs. In the source UTCT tables, the associations are implemented using foreign keys, in the traditional way. The mapping between the two schemas is discussed in the next section. In the archive version, associations are also implemented as foreign keys, but the key values are LSIDs. Even within our single collection of data we are beginning the creation of complex distributed data structures, linked by LSID.

4 Method

4.1 Specifying the Structure and Content of Archived Data

Our premise is that adding LSIDs to an existing database means declaring the structure of the

data to be archived as an *export-schema,* and specifying a mapping of the contents of an existing database to the export schema. This function is commonly fulfilled by SQL view definitions [13]. Thus, we define a domain specific language that is the syntactic subset of SQL used for such view definitions.

In our architecture, new tables are created from the view definitions. These tables are segregated from the run-time database and archived data is copied *(materialized*) into these tables. The additional tables and redundant storage of data greatly simplifies the implementation of a number of requirements of the LSID protocol. Perhaps the single most important simplification is that replication of archived data to multiple resolution services can be implemented using existing database replication products.

Figure 3, illustrates the LSID view definitions for two of the three records in our prototype. Due to its simplicity, the SpecimenImage view will be our primary example. The declaration provides that two attributes of each row of the Specimen table will be archived and assigned an LSID.

The second view definition, DarwinCoreRecord, is more complicated, but reveals the power of the approach with respect to its accessibility to biology labs. In UTCT, the data for a standard GBIF Darwin Core record comes from three sources. UTCT tables Specimen and Scan_Info are two of the sources. The individual fields of the Darwin Core record are populated using a conventional select/from/where query. Note, the AS clauses map UTCT attribute names to the names specified by the GBIF standard. String constants in the record are specified as well. Thus, any SQL programmer already familiar with a database, (e.g. the DBA), may make quick work of defining the archived portions of the database.

Besides inventing a new reserved word, LSIDVIEW the specification language changes SQL syntax in only one other place. The first line of the select clause for the Darwin Core Record specifies, "SpecimenImage.LSID AS GlobalUniqueIdentifier". SpecimenImage.LSID refers to the LSID for the archived version of the row in the specimen table. The expression is out of scope, referring to an attribute of a table in the archival database, and since the table was not declared in the FROM clause it is also syntactically incorrect. Since the statement is unambiguous, the semantics of our language already depart from SQL semantics, and the cure is worse than the defect, we leave well enough alone. When necessary, refer to our declarations as SQL-like and not SQL.

```
CREATE      LSIDVIEW SpecimenImage AS
SELECT      scientific_name, specimen_image
FROM        Specimen
```

a) An LSIDVIEW definition specifying a subset of the columns of a single table as the contents of a table in the export schema

```
CREATE      LSIDVIEW DarwinCoreRecord AS
SELECT      SpecimenImage.LSID      AS    GlobalUniqueIdentifier
            scan_date               AS    DateLastModified,
            'ct dataset'            AS    BasisOfRecord,
            'utexas'                AS    InstitutionCode,
            'utct'                  AS    CollectionCode,
```

```
            specimen_id             AS    CatalogNumber,
            scientific_name         AS    ScientificName,
           'http://utct-test.tacc.utexas.edu/data.php?specimen_id='          +
            specimen_id             AS    ImageURL,
            ''                      AS    RelatedInformation
FROM        Specimen, Scan_Info
WHERE       Specimen.id = Scan_Info.specimen_id
```

b) An LSIDVIEW definition detailing the mapping of records in two database tables to a single record defined by the Darwin Core standard and compliant with the standard, including, the LSID of the specimen record linking in additional data about the specimen. In effect, even within a standalone export schema LSIDs serve as foreign keys.

Figure 3: Sample LSIDVIEW Definitions

The differences in the semantics of our view definitions from SQL are required by the LSID semantics of versioning. In our system, views are always materialized and an LSID assigned to each record. Records are never removed from the materialized view. Inserts into the base tables may materialize additional records in the export database. Updates to the base tables may propagate new versions of the data. In that case, the assignment mechanism must recognize the update as such, reuse an existing LSID, but increment a version number.

We implement LSID assignment using triggers. Triggers are SQL procedures that are executed each time a table row is inserted or updated. It follows that *on-insert* triggers create new archived data, *on-update* triggers usually create new versions. The use of trigger-based methods to maintain the contents of materialized views after updates to base tables is well understood [3]. The LSID requirement that old versions persist simplifies these methods, as one need not determine which records may have to be removed.

If the database is a proxy for a laboratory notebook and only serves as an archive for voluminous output of high-throughput biology, we can exploit a simplifying assumption; a correction in the data means a change in the process which results in an entirely new data set. Thus, data is write-once and need only consider *on-insert* triggers. Such an implementation need simply, on each insert, generate an LSID string, write it and a copy of the archived data to the export database.

Updates, if allowed, are more complicated. The LSID for the prior version must be determined and reused. Also, sometimes data is accumulated in the database and not archived until it is "complete". If so, an update may simply mark data as complete and instigate the first and possibly only insertion of the data into the archive. Of the first, this is the case and will be detailed for the Darwin Core record defined in Figure 3b.

4.2 Architecture

There are three main components of the system; Figure 4. A compiler translates the external schema declaration into a set of triggers that monitor the database and implement the assignment service. The compiler also generates the schema for a set of tables that are populated by the triggers, and implementing dynamic storage needed for the protocol. Note, the existing database catalog is also an argument to the compiler as data types and related constraints must be included as part of the definition of the additional tables.

A fixed runtime system, whose only database parameters are the additional tables and their contents, implements the resolution service.

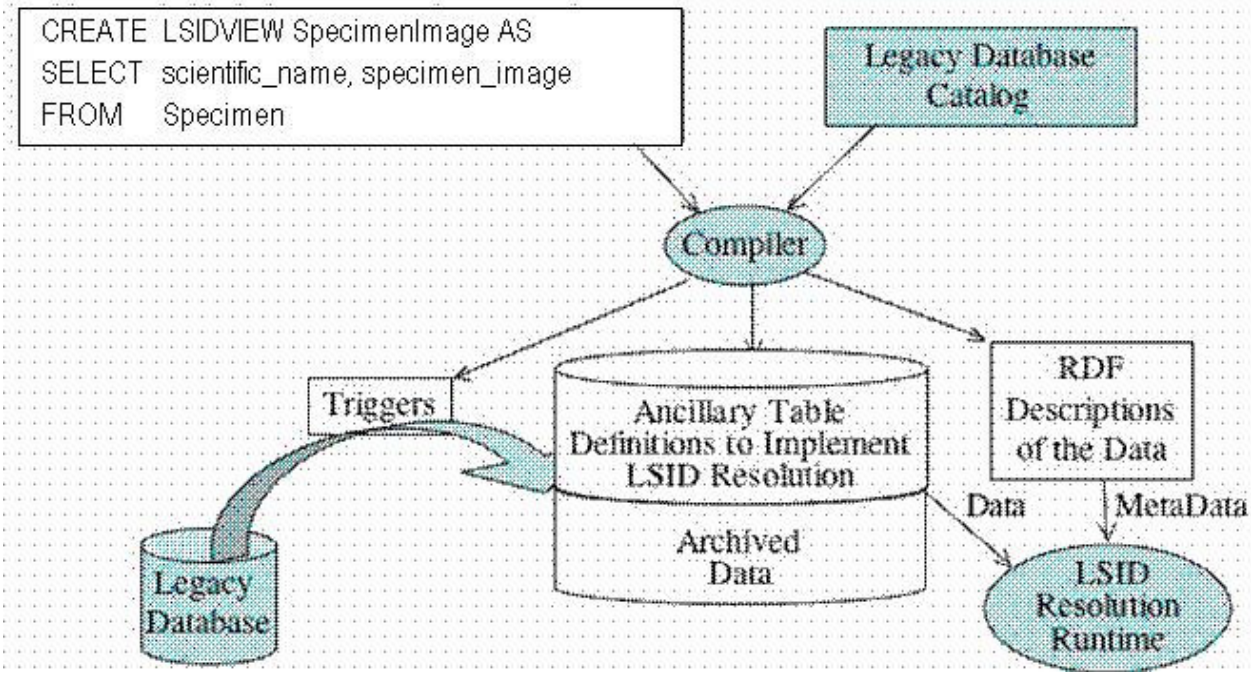

Figure 4: Domain Specific Language Based Implementation of LSIDs

4.3 Supporting Table Structures and the Semantics of Versioning

Most of the persistent state of the LSID implementation resides in database tables. A master table contains a complete list of LSID object identifiers, (lsObjID). An attribute of the master table indicates which table contains the data for a particular LSID. The indirection supported by the master table hides the internal structure of the data storage. The LSID specification stipulates that the LSID string obscure the internal mechanisms of the LSID authority. An alternative would be to include the table name in an internal version of the LSID string and then use an encrypted version of that string as the "official" representation of the LSID.

Two tables, *data* and *key,* are created for each LSIDView. Archived data is materialized in the data tables with the associated lsObjID and a version number. The lsObjID and version number form the primary key. The second table, the *key* table, has the same primary key as the data table, but contains the primary key value of the archived record. The data in these two tables is one-to-one and in principle there should only be one table. By organizing it as two tables, the data table contains precisely the archival record. The key table enables a vendor independent method for on-update trigger to determine a record's value, prior to update. The tables generated for the LSIDView in Figure 3a are illustrated in Figure 5.

LSIDTable	
lsObjID	Primary key
tableName	SpecimenImage / Other View names

SpecimenImageKey	
lsObjID	Primary key
version	
origPK	Primary key of original data, required for versioning.

SpecimenImageData	
lsObjID	Primary key
version	
scientific_name	Attribute from original database
specimen_image	Attribute from original database

Figure 5: The LSIDTable is part of the runtime system and is common and fixed to all implementations. Two additional tables, SpecimenImageKey and SpecimenImageData correspond to the implementation of the LSIDVIEW of Figure 3a. A copy of the exported specimen data will be maintained in the data table. The key table enables the implementation to avoid some vendor specific implementation details of triggers. By segregating the archived data from the ancillary information used to implement the protocol, commodity database replication software can be used to maintain consistency among multiple resolution services.

4.4 LSID Assignment as Triggers

We will first consider the simple case that an LSIDView contains a subset of the attributes of a single table. See Figure 3a. The triggers become more complicated when the views span multiple tables, a database join. See Figure 3b. In both cases, the greater challenge is in monitoring table updates and correctly creating new records using an existing LSID with an incremented version number.

4.4.1 Assigning an LSID for Single Tables

In this, the simplest case, as typified by the LSIDView in Figure 3a, a trigger is evaluated each time a row is inserted into a table. Pseudo code for the on-insert trigger:

1. Initialize a new LSID identifier string, lsObjID
2. Insert a record in the LSID table comprising the pair lsObjID and the archived table name, SpecimenImage
3. Copy the data to be archived by inserting a record into the SpecimenImageData table**.** In addition to the archived data, the record contains the lsObjID, the version number, which is one, and any additional attributes that form the primary key of the original table
4. Insert an entry in the SpecimenImageKey table for the new lsObjID and the original primary key of the data

The identification and replication of attribute values to be archived is accomplished by macro-

substitution of the select/from/where query in the LSIDView definition into the body of the trigger. The additional attributes are integral to the implementation of versioning, which happens only on updates. Pseudo-code for the trigger is illustrated above. A full implementation comprises some vendor specific code, particularly in regard to determining the value of the primary key. Our first implementation has been for SQL Server 2000. Appendix 1 contains the full SQL Server definition of this trigger. An ancillary paper contains complete the complete trigger definitions for the prototype [6].

4.4.2 Updating and Maintaining Versions for Project Only Views

The single challenge in versioning updates is to determine and reuse the LSID for the prior version. Once again, the issue is that, nominally, the scope of the trigger is within the original database and the LSIDs are only present in the archival database. The solution is the *key* tables in the archival database. The first operation of the on-update triggers is, using the primary key values of the updated row, prior to update, query the respective key table to determine the existing LSID. Pseudo-code for the on-update trigger:

1. Extract the primary key of the updated tuple and locate the latest version in the supporting table, identified by its LSID and version number
2. Extract attributes (per the view definition) from the updated tuple
3. Store a new copy of these attributes with the LSID and incremented version number in the supporting table defined for the view
4. Store an entry in the key table for the new copy

4.4.3 Complex Cases

There are two features that may complicate the LSIDVIEW definitions and implementation as triggers. First, it may not be desired that all rows in a table be made available for archival purposes, or there may be reason (e.g. pending publication) to delay public access to archive data. Second, archived data may be sourced from more than one table. It may simply not be possible to create the archive version of the record until the last of a number of inserts is processed across a number of tables. In the example of the UTCT both the Specimen record and the supporting CAT-scan slices must be in the database in order to form the Darwin Core record, Thus, these two issues are not separable.

In the situation represented by the UTCT Darwin Core record, correct behavior can be implemented by correctly defining the triggers, the details of which, though not trivial, have been understood for sometime [4]. If data is to be made available pending publication and there is provision in the existing database to record the publication information, the LSIDVIEW definitions can be conditioned on the insertion of the publication information. As in the UTCT Darwin Core record, until the publication information is inserted, the predicates making up the triggers could remain unsatisfied and no records copied into the archive. It is also possible a DBA could add a column to a table, call it *'archiveFlag'*, and only when the flag is set will the row be archived. Figure 6 illustrates how that would change the definition in Figure 3a.

```
CREATE      LSIDVIEW SpecimenImage AS
SELECT      scientific_name, specimen_image
```

```
FROM        Specimen
WHERE       Specimen.archiveFlag = TRUE;
```

Figure 6: The LSIDVIEW definition of Figure 3a augmented with an explicit test for archiving a row.

Pseudo code for on-insert trigger on an LSIDView involving more than one table:

1. Execute a join on the tables involved in the view, using only the inserted tuple in place of the entire table
2. For each tuple in the join:
    a. Initialize a new LSID
    b. Insert a record in the LSID table comprising the pair lsObjID and the archived table name
    c. Copy the data to be archived by inserting a record into the supporting table defined for the view
    d. Insert an entry in the key table

4.4.4 Updating and Maintaining Versions for Complex View Definitions

The updates defined on complex views are similar to those for similar views, with the exception that a single record is not updated, but all records in the join affected by updated record need to be re-versioned. Pseudo code of on-update trigger for complex views:

1. Execute a join on the tables involved in the view, using only the updated tuple in place of the entire table
2. For each tuple in the join:
    a. Extract the primary key of the updated tuple and locate the latest version in the supporting table, identified by its LSID and version number
    b. Extract attributes (per the view definition) from the updated tuple
    c. Store a new copy of these attributes with the LSID and incremented version number in the supporting table defined for the view
    d. Store an entry in the key table for the new copy

5 LSID Resolution and Meta-Data Syntax

The LSID protocol does not stipulate a preferred structure or syntax of the data or the metadata. In fact a single resolution service may provide both the data and metadata in more than one representation. The protocol specifies methods that enable clients to query a resolution service, asking which representations its supports. A client chooses among those to define how the information it receives is served.

There are many technologies competing to become *the standard* for Internet data exchange. These include, XML with or without schema definitions expressed as document data types (DTDs) or XML schemas (XSD), and the stack of representations connected with the semantic web, meaning RDF to OWL.

As this architecture speaks to relational data representations, a kind of least-common denominator, mapping the data and its metadata to any and all of these representations is

possible. The architecture specifies that data retrieval methods can return metadata in different formats, depending on the 'accepted_formats' parameter [1]. Alternately, abstract LSIDs can be used to support multiple representations [2].

6 Implementation Status and Discussion

We have a live implementation of the assignment and resolution services [10]. We chose the UTCT database as the archive and defined three LSIDViews. We then created a set of tables and triggers based on the view definitions. The triggers act as the LSID assigning service. An LSID resolution service has been written in PHP [11] that internally uses the LSID table definitions to compose the results for getData () and getMetadata () queries.

While we envision, given community interest, creating a compiler based implementation, the current implementation represents a first effort hand compiled implementation and a feasibility prototype. Among the three LSIDViews the trigger contents are vendor-specific but repetitious. We found the SQL Server implementation details straightforward. Thus, we deem compiler-based implementation is feasible.

The metadata and data, stored in relational databases are wrapped in RDF and then returned to the clients. This translation of database schema, for metadata and relational data to RDF is hand-written, based on Relational.OWL [5]. The implementation of a translation engine is underway.

Note that with the use of triggers, any update to the data will induce a new version consistent with LSID versioning semantics. This assumption may be relaxed if the DBA can develop a predicate that is satisfied when, and only when data is ready to be committed to a permanent public archive. If not, the system will still function but our claim that the system can be implemented on a legacy database without any alteration of the legacy system may no longer apply; the DBA may have to add a column to flag the intention to commit the record to the archive and the trigger can check this flag before executing any action.

7 Conclusion

We have defined a domain-specific SQL-like language to implement LSIDs for existing data stores. The language constructs can be used to define an export schema for a database detailing the objects that need LSIDs. Given these view definitions and the database catalog, a compiler can generate the required ancillary tables and triggers. The triggers are executed on the current data (archive) to assign LSIDs and persist the data versions. With this, we have defined an easy mechanism to implement LSIDs.

Acknowledgements

This research is funded by the National Science Foundation grant IIS-0531767.

References

[1] Life sciences identifiers specification. http://www.omg.org/docs/dtc/04-05-01.pdf

[2] Smith, D. and Szekely, B. LSID best practices. http://www-128.ibm.com/developerworks/opensource/library/os-lsidbp/
[3] Adelberg, B., Garcia-Molina, H., and Widom, J. The STRIP rule system for efficiently maintaining derived data. *Proceedings of the ACM SIGMOD International Conference on Management of Data,* 147-158, Tucson, Arizona, May 1997
[4] Garcia-Mollina, Ullman and Widom. Database Systems: The Complete Book. *Prentice Hall*
[5] de Laborda, C. P. and Conrad, S. 2005. Relational.OWL: a data and schema representation format based on OWL. *2nd APCCM2005, Vol. 43*
[6] Bafna, S. and Miranker, D. LSID Specification Language for Relational Databases based on SQL Server 2000. The University of Texas at Austin, Department of Computer Sciences, Technical Report *TR-06-51*, October 16, 2006
[7] Holmes, F. Reworking the Bench. *Springer*
[8] http://utct.tacc.utexas.edu
[9] http://digimorph.org
[10] http://utct-test.tacc.utexas.edu/authority
[11] http://biodiv.hyam.net/authority/readme.html
[12] http://darwincore.calacademy.org/
[13] http://www.w3schools.com/sql/sql_view.asp

Appendix

```
CREATE      TRIGGER PopulatePersistentSpecimenTables
ON          Specimen
INSTEAD OF  INSERT
AS

BEGIN

DECLARE     @lsObjID          INT
DECLARE     @newTuplePK       INT
DECLARE     @scientific_name  VARCHAR(256)

/* Get primary key of the tuple in newTuplePK */
SELECT      @newTuplePK=id
FROM        INSERTED

/* Get maximum lsObjID from LSID table */
SELECT      @lsObjID=MAX(lsObjID)
FROM        LSIDTable
IF          @lsObjID IS NULL
SET         @lsObjID = 0

/* Insert a new tuple in SpecimenImageLSIDTable with incremented lsObjID,
version number 1, original PK and new attribute values) */
SET         @lsObjID = @lsObjID + 1

SELECT      @scientific_name=scientific_name
FROM        INSERTED
```

```
INSERT      INTO SpecimenImageLSIDTable
SELECT      @lsObjID, 1, @newTuplePK, @scientific_name, specimen_image
FROM        INSERTED

INSERT      INTO LSIDTable
VALUES      (@lsObjId, 'SpecimenImageLSIDTable')

/* Insert new tuple in the Specimen table. Required because of use of instead
of trgiger, in order to support images */
INSERT      INTO Specimen
SELECT      *
FROM        INSERTED
END
```