DBTOOLS: A SUITE OF TOOLS FOR MANIPULATING INFORMATION IN A RELATIONAL DATABASE

Sanford Ballard and Jennifer Lewis

Sandia National Laboratories

Sponsored by National Nuclear Security Administration
Office of Nonproliferation Research and Engineering
Office of Defense Nuclear Nonproliferation

Contract No. DE-AC04-94AL85000

ABSTRACT

While loading data into a database is a simple operation, doing so *intelligently* is deceptively difficult. Intelligent loading means loading data in a manner that, at a minimum, (1) does not duplicate information already in the database, (2) links new information to related information already in the database, and (3) remaps identification numbers in the input data to prevent conflict with identification numbers already in the database. In this paper, we present a suite of tools that we have developed to help users intelligently manipulate database information.

- EvLoader – This tool merges one or more rows from a source EVENT table into a target EVENT table. All information in the database that is linked to the source event row(s) is also merged (the user specifies what information is related to the source event in the parameter file). The code operates in two modes. In the first, origins in the source event are merged with origins in the target event based on EVID number. In the second mode of operation, source events are merged with target events based on spatial/temporal correlation. Origins that are members of the same Event in the source tables will remain members of the same Event in the target tables.

- Unloader – This tool deletes one or more specified rows from the database and also deletes all rows that are connected to the specified row(s) according to relationships specified in the Entity Relationship Diagram. Rows are only deleted if doing so will not violate any foreign key relationships in the database.

- FileInterface – This tool recognizes three information storage formats: a set of database tables, a set of ASCII flatfiles, and XML. The tool can convert information from any one of these formats into any of the others. For the XML format, we have designed an XML schema document that completely captures all of the information in a data set.

- Remap – Given two sets of similar database tables, this tool will generate a remap table which will relate the identification numbers in the source tables to the identification numbers in the target tables.

All of these tools depend on a library of utilities called DBToolBox. DBToolBox is written in a completely generic manner in the sense that it does not have hard-coded within it any information about any particular set of tables, column descriptions, or entity relationship diagram. All such schema-specific information is supplied by the user via parameter files and Table Definition Tables. This approach makes the tools that use DBToolBox immune to most schema changes such as addition/deletion of columns from a table or changes in the size of a particular data element (i8 to i9, for example).

OBJECTIVES

Many of the custom software tools that have been developed to support nuclear-explosion monitoring do not insert the results of the calculations that they perform back into the relational database where data are stored. This is due primarily to concern that the new data will be inserted incorrectly, which might not only make the new data inaccessible, but may also corrupt data that already reside in the database. To remedy this situation, we have developed a library of software tools, called DBToolBox, that facilitates proper insertion of new information into a relational database. DBToolBox, which is written in the Java programming language, can be included in software applications and allow those applications to insert, update and delete rows of information in the database while ensuring that new identification numbers (IDs) do not conflict with existing IDs, that foreign key relationships are obeyed (even without having foreign key constraints turned on in the database) and that unique keys are honored.

Besides the DBToolBox library itself (a Java package), we have developed a number of applications built on DBToolBox. Each of these applications will be described below.

RESEARCH ACCOMPLISHED

DBToolBox uses graph theory (Sedgewick, 2004) extensively in its implementation. A graph is a mathematical concept involving a collection of vertexes and another collection of edges, which are connections between pairs of vertexes. A directed graph is a special type of graph in which all of the edges have a direction, i.e., they originate in one vertex and terminate in another. A set of tables in a database constitute a directed graph, which we will refer to as a TableGraph, where the tables are the vertexes and the foreign key relationships are the edges. An entity relationship diagram of the tables in a relational database is a representation of a TableGraph. Consider Figure 1, which illustrates the entity relationship diagram for the core tables of the NNSA schema (Carr, 2002). Each box in the diagram represents a table (vertex) in the TableGraph and each line connecting two tables represents two edges in the TableGraph, one going in each direction. The edges can be expressed as SQL select statements that include where clauses involving foreign keys. For example, consider the line between the Origin Table and the Event table that is labeled orid-prefor. Given a row from the Origin table (which will have a unique orid value), one can represent the edge in the TableGraph that extends from the Origin table to the Event table with the select statement "select * from Event where prefor=#orid#' where #orid# represents the actual value of the orid field in the current row of the Origin table. The select statement 'select * from Origin where orid=#prefor#' represents an edge in the TableGraph that originates in the Event table and terminates in the Origin table.

There is another, even more useful graph in a database which we will call a RowGraph. A RowGraph is a directed graph in which the vertexes are rows from the database and edges emanate from a row that produced another row to the row that was produced. A RowGraph is constructed by starting from one or more rows in the database and executing all the edges in the TableGraph that emanate from the table of which the row is a member (execution of an edge in a TableGraph means execution of the SQL statement that the edge represents). All of the rows produced by the execution of the edge are added to the RowGraph, provided that the RowGraph does not already contain that row. For each row that is added to the RowGraph an edge is added that originates in the row that produced the new row and terminates in the new row. When each new row is added to the RowGraph, all of the edges in the TableGraph to which the row belongs are executed and the new rows produced are also added to the RowGraph. This process continues until no more rows are added to the RowGraph. Note that it is critical that rows only be added to the RowGraph once. If this precaution is not adhered to, it is possible to enter an infinite loop.

From the point of view of application development, a RowGraph is an extremely useful construct. Once a set of database tables has been identified and the relationships between those tables defined, one may specify some initial row in the database and extract all the other rows in all the other tables in the database which are related to the initial row by one or more of the defined relationships. If the tables and relationships are properly defined, then the application will have in memory all the rows that it needs to do its work, and no more. Not only that, it will also have the information needed to efficiently traverse the data set and visit the rows with which it needs to interact. At the most abstract level, the role that DBToolBox performs is to manage the assembly, traversal, modification and input/output of RowGraphs.

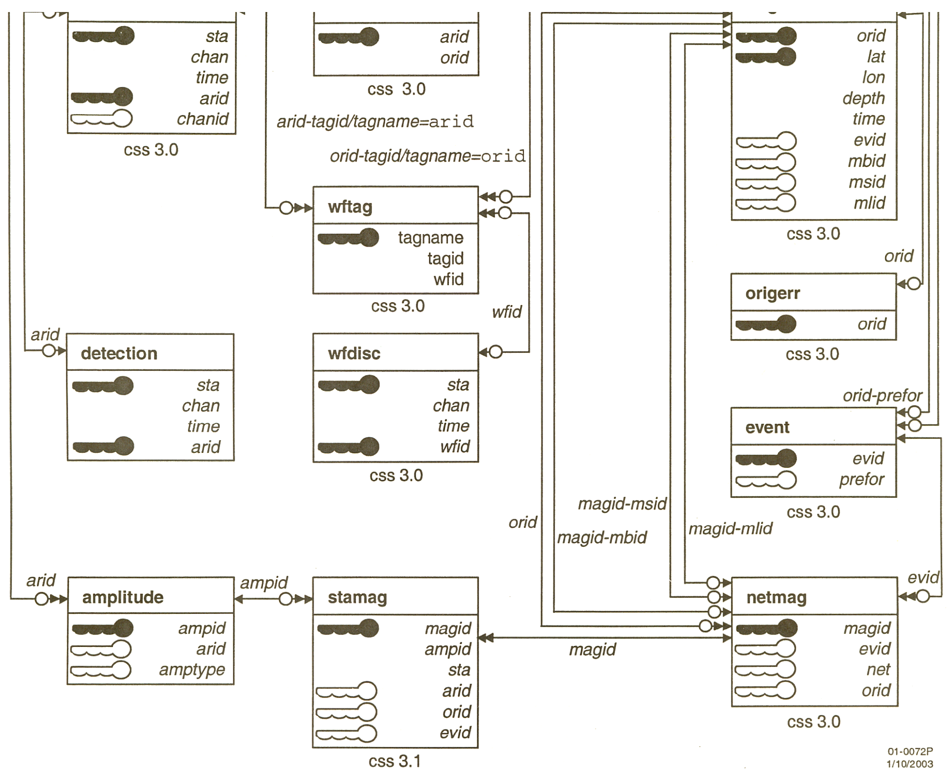

Figure 1. TableGraph of the tables and relationships in the NNSA core schema.

System Architecture

DBToolBox is a Java package that instantiates objects from a number of classes. The primary classes can be divided into two major categories: those that encapsulate specific features or properties of the database system and those that implement specific functions that operate on the system.

The major property-type classes managed by DBToolBox are organized as illustrated in Figure 2. At the top of the structure is the RowGraph object, of which there is generally only a single instance in a DBToolBox session. The RowGraph object contains one or more Schema objects, each of which encapsulates the features of a TableGraph as described in the previous section. A Schema object has a collection of Table objects, which are the vertexes of the TableGraph, and a collection of Relationship objects, which are the edges of the TableGraph. Relationship objects manage the execution of SQL select statements between an origination table and a termination table.

Each Table object knows its name (the name of the table in the database), and what type of table it is (as defined in the database schema documentation). Each Table maintains a collection of Column objects, each of which knows the name of the column in the database table, its SQL type, NAValue, foreign key status, etc. Note that Schema, Table and Column objects do not contain any data from the database, per se, but rather serve to manage all the metadata that describes the actual data.

Actual data are contained within Row objects, each of which has a collection of data values and a reference to the Table object that contains the Row. The architecture was designed this way because typical applications will likely only have to deal with a limited number of Schema, Table and Column objects but may have to instantiate a very large number of Row objects. Given this consideration, Rows are designed to have the smallest memory size possible while all the metadata describing the data in a Row object is accessible through a single reference to a Table object. A Row object also has a unique identifier, called a RowID, which is used to identify the Row in the RowGraph of which it is a member. The RowID is computed as the MD5 hash (Rivest, 1992) of a string

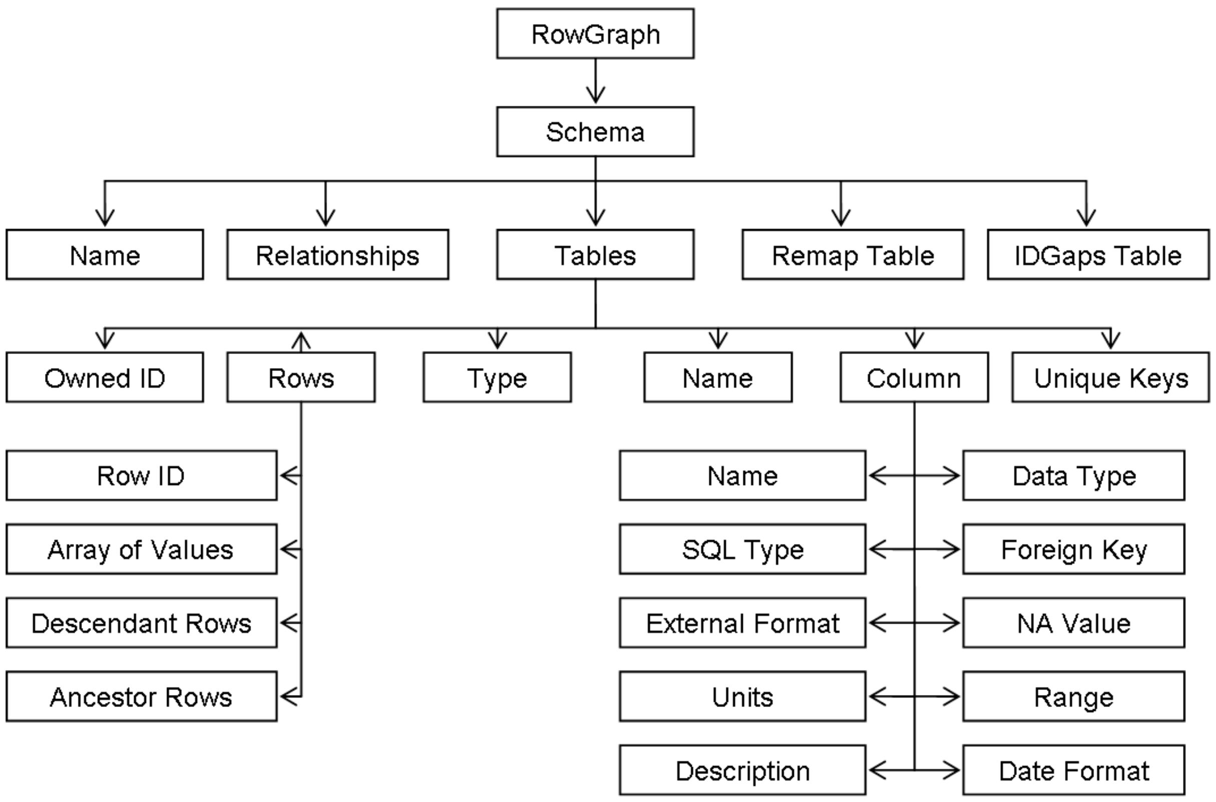

Figure 2. DBToolBox system architecture.

constructed from the following values: the name of the Schema object that the Row is a member of, the type of the Table object that the Row is a member of, and string representations of all the data values in the Row, with the exception of LDDATE (the date that the information was loaded into the database). It is imperative that the RowID be unique so that the same row of information will not be added to the same RowGraph more than once.

The only object in the system that contains a collection of Row objects is the RowGraph object. In addition to the Row objects, a RowGraph keeps track of all the edges in the RowGraph. A RowGraph is designed so that it can efficiently retrieve all the Row objects that come from a particular Schema or Table object, all the direct descendents of a particular Row, all the direct ancestors of a particular Row, and more. RowGraph implements the Java Set interface, so it has iterators and all of the other features of a Set that Java developers are familiar with.

Several classes in DBToolBox manage functional behaviors rather than object features and properties. To describe how these operators work, it is necessary to clearly define a number of terms as they are used in the context of DBToolBox operations:

ID owner table – DBToolBox distinguishes between two fundamental types of tables, those that 'own' an ID and those that do not. Each ID must be 'owned' by one and only one table. ID owner tables tend to be tables that represent some fundamental quantity that is of interest to the users of the database. In the NNSA core schema, the tables that are ID owners include Event, which owns evid, Origin (orid), Arrival (arid), and Netmag (magid). Non ID owner tables tend to be tables that contain supporting information (Origerr for example), or tables that serve as a bridge between two ID owner tables (Assoc for example). Non ID owner tables contain fields that relate to fields in ID owner tables, forming foreign key relationships with those tables.

Unique keys – these are fields in a row of a table that, taken as a group, should not be duplicated in the table. An important corollary is that two rows that come from different tables of the same type and which have equal unique keys, are considered equal, regardless of the values in the other fields of the two Rows. This means that if a source row is to be added to a target table but a row already exists in the target table with equal unique keys, then the source row need not be added to the target table but all other source rows which are related to the discarded source row by foreign key relationships may be linked to the target row that has unique keys equal to the discarded source row. As used in DBToolBox, unique keys may include only data fields and must not include any IDs. In the NNSA schema, many of the fields defined as primary keys are used in DBToolBox as unique keys. For example, the NNSA schema defines the Origin table to have primary keys LAT, LON, DEPTH, TIME and AUTH. In DBToolBox, these would constitute a set of unique keys.

The most important of the functional type classes is the Merge class, which implements methods to merge a collection of Rows (which are linked together in a RowGraph), into a set of target Tables where the target Tables are not the same tables as the ones from which the Rows originated. Merging information from one set of tables into another set presents a number of challenges. The first is that the IDs in the source Rows may conflict with IDs in the target Tables. To overcome this difficulty, the IDs in the source Rows are renumbered using new ID values that do not currently exist in the target Tables. DBToolBox has an IDGaps class to facilitate this renumbering (IDGaps keeps track of used id values in a set of tables, including gaps in the ID values). Another challenge when merging Rows into a different schema is that duplication of unique keys in the target table is to be avoided. Before inserting a Row from the source RowGraph into the target Table, Merge checks to see if a Row with equal unique keys already exists in the target Table. If one or more such Rows are found, Merge will refrain from duplicating the information in the target Table and will renumber IDs in the source Rows so as to preserve the necessary foreign key relationships.

The Merge operation executes three passes through the collection of source Rows that are to be merged into the target tables. During the first pass, only Rows that come from Tables that own an ID are visited. Each Row from an ID owner Table is examined to see if a Row exists in the corresponding target Table that has equal unique keys. If such a Row is found then an entry is made in the Remap Table that relates the value of the owned ID in the source Row to the value of that ID in the target Row, and the source row is labeled in such a way that it will not be copied into the target tables. If no target Row with equal unique keys is found, a new value for the owned ID is retrieved from the target IDGaps Table, an entry is made in the Remap table that relates the value of the ID in the source Row to the new value obtained from the IDGaps table, and the Row is labeled such that it will be copied into the target tables. Other than the labels, the source Row is not modified during this first pass.

During the second pass, Merge visits only Rows from non-ID owner tables. All the IDs are renumbered using the remap relations established during the first pass of the Merge operation. Then the target Tables are searched for a Row that has equal ID values. If none are found, the non-ID owner source Row is labeled for insertion into the target Tables. If one is found, it must be true that all the ID owner source rows with which the non-ID owner source row shares foreign key relationships found target rows with equal unique keys and were labeled such that they would not be copied into the target tables. In this case, if unique keys were defined for this non-ID owner table, the non-ID target row is checked to see if it has unique keys that are equal to those in the source Row. If unique keys were defined, and the unique keys in the target Row are not equal to those in the source Row, there is a conflict. The ID owner rows that the non-ID owner Row is related to are not to be copied into the target table but will have their IDs remapped to ID values that already exist in the target table. In order to avoid duplication of the primary keys in the non-ID owner Row, the non-ID owner source Row should not be copied into the target tables either. But, since the unique keys have unequal values, critical information will be lost (it is assumed that unique keys contain critical information). We are confronted with a dilemma. On the one hand, we do not want to duplicate unique keys in the ID owner Rows but on the other hand we do not want to lose critical information contained in the non-ID owner Row. The DBToolBox Merge function will always choose to duplicate information in one of the ID owner tables rather than lose critical information in the non-ID owner table. When confronted with this situation, it will reverse a decision made during the first pass through the RowGraph where it decided to not copy a Row of information to the target Tables.

During the third pass through the RowGraph, all Rows that were labeled for insertion into the target tables are visited. A new Row is created that honors the target Schema. ID fields in the source Row are renumbered using the Remap information generated during the first two passes through the RowGraph and copied into the target Rows. Then all the data fields are copied from the source to the target Row and the new target Row is inserted into the target Table.

Applications

The DBToolBox library of utilities described in the previous section can serve as the basis for a variety of applications that need to interact with a database. So far, we have written four applications that perform system management activities. These applications are described below. In addition to these applications, DBToolBox is used by the Peer to Peer Tool Communication (Merchant et al., 2004). RowGraphs serve as the fundamental objects that are passed around during interprocess tool communication. While DBToolBox is written in a completely generic manner in that no specific schema information such as table names or column definitions are hard-coded into the library itself, that is not true of the applications described below.

EvLoader

This tool merges one or more rows from a source EVENT table into a target EVENT table. All information in the database that is linked to the source event row(s) is also merged (the user specifies what information is related to the source event in the parameter file). EvLoader uses DBToolBox to build a RowGraph starting from the initial EVENT rows specified by the user. It then invokes the Merge function in DBToolBox to merge the source rows into the target tables in such a way that primary keys, foreign keys and unique keys are properly managed.

The code operates in two modes. In the first, origins in the source event are merged with origins in the target event based on EVID number. In this mode, EVID numbers in source rows are not renumbered but are inserted into the target tables without modification. This mode of operation is used by AFTAC to merge data from the various pipeline database accounts (SOCPRO, AL1, AL2, EVAL), into the AFTAC global account.

In the second mode of operation, source events are merged with target events based on spatial/temporal correlation. The code is used in this mode to merge global and regional seismic event catalogs into a single database. The goal is to combine origins from disparate sources into coherent events based on the spatial-temporal proximity of the origins in the events.

A precise description of the algorithm used by EvLoader is provided below, using set notation.

Given:

1. a single source event consisting of 1 or more origins
2. a target event table where each row represents an event consisting of 1 or more origins,
3. no two origins that are members of the same event may have the same author. This must be true for the source event and must be true for all events in the target event table, both before and after merging of the source event.
4. Source origins which are ‘equal’ to a target origin that already exists in the target origin table are not duplicated in the target origin table. Origin equality is defined as:

   ```
   where source.auth=target.auth
   and abs(source.lat-target.lat)<1e-3
   and abs(source.lon-target.lon)<1e-3
   and abs(source.depth-target.depth)<1e-2
   and abs(source.time-target.time)<1e-3.
   ```

5. Source origins that were all members of the same source event prior to the merge will all be members of the same target event after merging. Integrity of target events is not guaranteed, however. Target origins which were ‘equal’ to source origins will be repointed from the target event of which they were a member before merging to the target event into which the source origins are being inserted.

Variables

The following variables are defined:

EventEqualityTest: A SQL WHERE clause that defines what it means for two events to be ‘equal’. Examples might include ‘WHERE EVID=#EVID#’ or ‘WHERE EVENT_NAME=#EVENT_NAME#’. To evaluate the EventEqualityTest, EvLoader will, for a given row from the source EVENT table, replace the field names enclosed in ‘#’ characters with the value of those fields from the source EVENT row and execute the select statement against the target EVENT table. If one target EVENT row is returned, the test returns the event that that row represents. If no rows are returned, the test returns $\varnothing$. If two or more rows are returned an error condition is asserted. The EventEqualityTest may be undefined.

E_t: The set of target events that are each a potential recipient of the source origins. The set is defined by a SQL select statement based on the contents of the preferred origin of the source event. The following example will select all target events whose preferred origin is within 30 seconds in time and 100 km in horizontal distance from the preferred origin of the source event:

```
select * from target_event_table where prefor in (select orid from
target_origin_table where abs(time-#time#)<30 and distance(lat, lon, #lat#,
#lon#) < 100)
```

The field names enclosed in ‘#’ characters are replaced with the values of those fields extracted from the preferred origin of the source event.

e_t : a single target event, which is defined in the algorithm. At the conclusion of the algorithm, e_t is the target event to which source origins will be added.

T : the set of origins in e_t.

T_{eq}: a subset of T such that for each origin in T_{eq} there exists an origin in S with the same author, lat, lon, depth and time. The subscript *eq* denotes ‘equality’.

T_a: a subset of T such that for each origin in T_a there exists an origin in S with the same author but with a different lat, lon, depth or time. The subscript *a* denotes ‘author’.

T_u: a subset of T such that for each origin in T_u there are no origins in S with the same author. The subscript *u* denotes ‘unique’.

t_p : the origin in T_u that has the highest author ranking. If T_u is empty, $t_p = \varnothing$. t_p is only guaranteed to be the preferred origin in e_t if $T_a \cup T_{eq} = \varnothing$, or stated another way, $T_u = T$.

S: the set of origins in the source event, with the constraint that $S \neq \varnothing$.

S_a: a subset of S such that for each origin in S_a there exists an origin in T with the same author but with a different lat, lon, depth or time. The subscript *a* denotes ‘author’.

s_p: the preferred origin in the source event.

S_g: the origins in S for which there exists an origin in the target ORIGIN table which is ‘equal’ to the source origin. The subscript *g* denotes ‘global’.

T_g: the origins in the target ORIGIN table for which there exists an origin in S which is ‘equal’ to the target origin. The subscript *g* denotes ‘global’.

E_g: the subset of all target events in the target EVENT table such that each event in E_g contains only origins in T_g. These events contain only origins that are equal to origins in the source event.

e_{best}: the event in E_g that contains the maximum number of origins. If more than one event in E_g contains the maximum number of origins, e_{best} can be any one.

Observations

$(T_u \cap T_{eq}) \cup (T_{eq} \cap T_a) \cup (T_a \cap T_u) = \varnothing$. This says that T_u, T_a and T_{eq} are each mutually distinct (they don't overlap).

$T_{eq} \cup T_a \cup T_u = T$ *which indicates that* T *does not contain any target origin that is not a member of either* T_u, T_a *or* T_{eq}.
S_g and T_g are the same size but may both be empty.
S_a and T_a are the same size but may both be empty.

Given the definitions above:
$(T_g - T_{eq})$ will include all target origins that are 'equal' to a source origin but which are not already members of the target event, e_t. These are origins in the target ORIGIN table which should be repointed to e_t.
$(S - S_g)$ will include all the origins in the source event, except those which are 'equal' to an origin that already exists in the target ORIGIN table. These are the source origins that need to be added to e_t.

Algorithm

1. If the EventEqualityTest is not defined proceed to step 2. Otherwise, define e_t to be the target event returned when the EventEqualityTest is evaluated.
    a. If $e_t \neq \varnothing$, goto 4.
    b. If the EventEqualityTest is "`where evid=#evid#`", define e_t to be a new target event with the same evid value as the source event. Goto 4.
    c. Goto 2.
2. If $T_u \neq \varnothing$ for any $e_t \in E_t$, find $e_t \in E_t$ such that t_p is closest to s_p
    a. If $T_a = \varnothing$, goto 4.
    b. If the separation between the centroid of S_a and the centroid of T_u is greater than the separation between the centroid of T_a and the centroid of T_u, goto 3.
    c. If the separation between the centroid of S_a and the centroid of T_u is less than the separation between the centroid of T_a and the centroid of T_u, create a new target event with a new evid from the ID_GAPS table and repoint the elements of T_a to it. Goto 4.
3. If $E_g \neq \varnothing$, set $e_t = e_{best}$. Otherwise, define e_t to be a new target event with a new evid from the ID_GAPS table. Goto 4.
4. Repoint $(T_g - T_{eq})$ to e_t and add $(S - S_g)$ to e_t. Terminate the algorithm.

Unloader

Deletion of a row from a database is a deceptively simple operation to execute in major database applications such as Oracle, but, when foreign key constraints are turned off in the database, as they frequently are, the unintended consequences of row deletion can be profound. Row deletion can break numerous foreign key relationships, which can severely compromise the integrity of the data in the database. With Unloader, the user specifies one or more rows from a single table that are to be deleted and also specifies a series of relationships that lead to more rows in other tables in the database. The intention is to delete all of these rows. Unloader takes this information and constructs a RowGraph based on the tables, relationships and starting rows specified by the user. It then analyzes the RowGraph to ensure that deletion of all these rows will not break any foreign key relationships. Finally, Unloader deletes from the database only the rows in the RowGraph that will not violate the integrity of other data in the database.

The operations performed by Unloader are perhaps best illustrated with an example. Consider an Event with two Origins, each of which is related to 5 Arrival rows through 5 Assoc rows (a total of 10 Assocs and 5 Arrivals). The user requests that one of the Origins be deleted (the one that is the preferred origin of the Event row), and specifies relationships from the Origin table to the Assoc and Arrival tables. Unloader constructs a RowGraph with the Origin that is to be deleted, the 5 Assocs that are related to the Origin and the 5 Arrivals that are related to the Assocs. It analyzes all the foreign key relationships involving these rows in the RowGraph. For the Origin row, it will consider the orid-prefor relationship with the row in the Event table and will modify the Event row since the Origin that is to be deleted is the preferred Origin of the event. If the Origin to be deleted was the only Origin in the Event, it would have deleted the Event as well. Next, it considers all of the Assoc rows and determines that all the rows with which the Assoc rows have foreign key relationships are themselves members of the RowGraph and that the Assoc rows can therefore be safely deleted. At that point it moves on to the Arrival rows where it finds that there are other Assoc rows, which are not members of the RowGraph to be deleted, that have relationships with the Arrival rows. It concludes that the Arrival rows may not be deleted. Finally, Unloader deletes the Origin row, the 5 Assoc Rows and modifies the Event row, but leaves the Arrivals intact.

FileInterface

Not all of the data that researchers must deal with are stored in the database. Data must frequently be imported from flatfiles, or must be exported to flatfiles so that it can be transferred to other users. Also, as mentioned earlier, some applications used by researchers write their output to flatfiles rather than to the database because of the difficulty of implementing database interactions. FileInterface facilitates the transfer of data between the database and flatfiles. It recognizes three information storage formats: a set of database tables, a set of ASCII flatfiles, and XML. The tool can merge information from any one of these formats into any of the others, using the DBToolBox Merge functions described earlier.

For the XML format, we have designed an XML schema that completely captures all of the information in a data set, including both the data itself and the metadata that describes the data. The schema closely mimics the architecture of the DBToolBox system illustrated in Figure 2.

Remap

The Remap utility generates a new remap table between two tables that purportedly contain the same data but with different ID values. This allows researchers to discover the identification numbers in the merged database that equate to their original identification numbers, even in the case where the Remap tables generated during merge operations are lost.

Remap operates as follows: For each ID for which remap information is to be generated, the user specifies a source table, a target table, and a SQL select statement that defines equality of rows in the source and target table. The specified tables should be the ones that ‘own’ the ID under considerations. For every row in the source table, the select statement is executed against the target table, and, if a single row is returned, an entry in the remap table is generated that relates the value of the owned ID in the source table to the value in the target table. If no row is retrieved, no entry is made. If more than one entry is retrieved, an error condition is asserted.

CONCLUSIONS AND RECOMMENDATIONS

We have developed a library of software modules called DBToolBox that facilitates interaction between software tools and data stored in a relational database. The modules are totally generic in that the specifics of the schema that describes the relational database that the code operates on are not hard-coded into the software but rather are stored externally in database tables or XML files. Schema modifications, additions and enhancements can be made to the external data structures and implemented without requiring modification to the DBToolBox library.

To date, we have developed four applications based on DBToolBox. These applications merge information from disparate databases into a single set of tables, safely delete information from a set of tables without compromising the integrity of the data that remains, convert data stored in database tables, flatfiles and XML files to and from each of these formats, and generate ID remap tables between compatible sets of database tables. In addition, DBToolBox RowGraphs serve as the fundamental data objects that are passed between software tools in the Peer to Peer Tool Communication System (Merchant et. al., 2004).

ACKNOWLEDGEMENTS

Many thanks to Chris Young for support and guidance during development of this code and for reviewing the manuscript.

REFERENCES

Carr, D. (2004), National Nuclear Security Administration Knowledge Base Core Table Schema Document, Sandia National Laboratories, SAND2002-3005, available at https://www.nemre.nnsa.doe.gov/prod/nemre/fileshare/coretables_062104.pdf.

Merchant, B. J., D. Hart, E. Chael, M. Chown, J. Hampton (2004), Proceedings of the 26th Seismic Research Review: Knowledge Base Navigator Facilitating Regional Analysis Inter-Tool Communication, Orlando, FL (Sept. 21-23, 2004).

Rivest, R. (1992), "The MD5 Message-Digest Algorithm," http://theory.lcs.mit.edu/~rivest/Rivest-MD5.txt.

Sedgewick, R. (2004), Algorithms in Java, Part 5, Graph Algorithms, 3rd Edition, Addison-Wesley.