SANDIA REPORT
SAND2007-5992
Unlimited Release
Printed September 2007

GNEMRE DBTools: A Suite of Tools for Access, Maintenance, and Manipulation of Data in a Relational Database

Jennifer E. Lewis & Sanford Ballard

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94AL85000.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd.
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online order: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2007-5992
Unlimited Release
Printed September 2007

GNEMRE DBTools: A Suite of Tools for Access, Maintenance, and Manipulation of Seismic Data

Jennifer E. Lewis & Sanford Ballard
Next Generation Monitoring Systems
Sandia National Laboratories
P.O. Box 5800
Albuquerque, New Mexico 87185-0401

Abstract

DBTools is comprised of a suite of applications for manipulating data in a database. While loading data into a database is a relatively simple operation, loading data intelligently is deceptively difficult. Loading data intelligently means: not duplicating information already in the database, associating new information with related information already in the database, and maintaining a mapping of identification numbers in the input data to existing or new identification numbers in the database to prevent conflicts between the input data and the existing data.

Most DBTools applications utilize DBUtilLib - a Java library with functionality supporting database, flatfile, and XML data formats. DBUtilLib is written in a completely generic manner. No schema specific information is embedded within the code; all such information comes from external sources. This approach makes the DBTools applications immune to most schema changes such as addition/deletion of columns from a table or changes to the size of a particular data element.

ACKNOWLEDGMENTS

We thank all of the DBTools users who have helped us to debug and improve the software, especially our colleagues at Los Alamos National Laboratories and Lawrence Livermore National Laboratories. In particular, we are grateful to Richard Stead for his development of the schema schema.

CONTENTS

FIGURES

TABLES

1. INTRODUCTION

DBTools was initially developed for two reasons. First, because the database content required to be integrated into the Ground-based Nuclear Explosion Monitoring Research & Engineering (GNEMRE) Knowledge Base (KB) had become too large and the associated schema too complex to be handled with simple SQL scripts. Second, because many of the custom software tools in use at the National Nuclear Security Administration (NNSA) laboratories did not have the capability to insert the results of the calculations they perform back into the relational database where data is stored. This was primarily due to the fact that inserting data into a relational database is non-trivial; consequently, many application developers avoid inserting data altogether rather than risk doing it incorrectly. Incorrectly inserting data can make the new data difficult to access, may corrupt existing data in the database, and may result in data duplication.

After researching existing software to assess if any existing software vendors had already implemented solutions to these issues, we developed DBTools. DBTools consists of an assortment of applications that perform a range of data manipulation functions (insertion, deletion, extraction, format conversion, merging) for data in a database, flat files, or XML format.

All of the DBTools applications build upon the functionality in DBUtilLib. DBUtilLib is a library of software utilities that facilitates:

- Proper insertion of new information into a relational database
- Proper removal of existing information from a relational database
- Proper merging of information into an existing relational database

DBUtilLib is written in Java [1] and can be included in software applications. It allows applications to insert, update, and delete rows of information in the database while ensuring that:

- New IDs (identification numbers) do not conflict with existing IDs
- Foreign key relationships are honored even when foreign key constraints are not enabled in the database
- Unique key constraints are honored even when unique key constraints are not enabled in the database

Throughout this document, database table names will be in **bold** and column names will be in *italics*.

The intent of this document is to describe what functionality the DBTools software provides. The intended audience is those who are interested in learning about this functionality. DBTools was primarily developed for seismologists and those performing research in the field of seismology. A User's Manual and sample parameters files are provided with the DBTools software.

This page intentionally left blank.

2. KEY TECHNICAL CONCEPTS

2.1 Table Definition Table

DBTools applications make use of a table definition table in order to obtain information about tables being processed. A table definition table is a collection of metadata about tables that is stored independently of the tables themselves and the code. For those that are familiar with the set of GNEMRE database schema tables known as the schema schema, the table definition table is a view created from two of these schema schema tables – **coldesdcript** and **colassoc**.

Having this table metadata information available externally prevents the code from needing any information (e.g. column names, column formats, primary key constraints, unique key constraints, etc.) about the tables embedded within the code itself. This allows the code to be highly flexible when table formats change. For example, when a column type changes from an i8 to an i9, no code needs modification and recompilation. The only change needed is within the table definition table for that column. After that change is made, the code will read in the updated metadata from the table definition table and handle the new format accordingly.

Table definition tables also contain information that can be used when parsing table data from a flat file. Thus, if the number of characters defined for a particular column within a particular type of flat file changes, no flat file parsing code must be changed. The change is instead made within the table definition table, which is used by the code when parsing flat files.

Another benefit to having this externally defined table definition table is that database tables can be created using the information in the table definition table. Typically, tables exist in the database and software acquires metadata about those tables from the database's data dictionaries. Using table definition table information, tables can be created with the correct column names, column formats, primary key constraints, and unique key constraints. Again, this makes the DBTools applications highly flexible in what they are able to achieve within the database.

Table 1 lists the columns in a database table definition table.

Table 1. Table Definition Table Columns

TABLE_NAME	Name of the type of table being defined.
COLUMN_NAME	Name of a column within the table defined in TABLE_NAME.
COLUMN_POSITION	Numerical "position" of the column within the table; (i.e., if COLUMN_POSITION is 1, then the column defined in COLUMN_NAME is the 1st column in the table defined in TABLE_NAME).
KEY	If the column defined in COLUMN_NAME is a foreign key, the KEY column contains the name of the column in some other table that the column defined in COLUMN_NAME refers to. If the column defined in COLUMN_NAME is the table's owned ID, the KEY column contains the value `ownedid!`.
EXTERNAL_FORMAT	The column defined in COLUMN_NAME's external format. This is the formatting applied to the column's value when writing that value to a flat file. Note that if this is a date, that date's formatting information must be a format that adheres to Oracle's standard for date formatting.
EXTERNAL_WIDTH	The field width of the flat file format for the column defined in COLUMN_NAME.
INTERNAL_FORMAT	The format for the column in the database for the column defined in COLUMN_NAME.
SCHEMA	The schema that table and column information is being defined for. (Different schemas may have different table and column information.)
NA_ALLOWED	A Boolean (y or n) to indicate whether or not an NA (not available) value is allowed for the column defined in COLUMN_NAME within the table defined in TABLE_NAME.
NA_VALUE	The NA Value for the column defined in COLUMN_NAME if an NA Value is allowed. NA values are used to indicate that information is not available for a column.
COLUMN_TYPE	The type for the column defined in COLUMN_NAME. This must be one of the following 6 values: primary key, unique key, foreign key, descriptive data, measurement data, administrative data.
EXTERNAL_TYPE	The external type for the column defined in COLUMN_NAME. This is how this column will be represented within memory by Java code.

Table 2 contains some sample data from a table definition table.

Table 2. Sample Table Definition Table Data

TABLE_NAME	COLUMN_NAME	COLUMN_POSITION	KEY	EXTERNAL_FORMAT	EXTERNAL_WIDTH	INTERNAL_FORMAT	SCHEMA	NA_ALLOWED	NA_VALUE	COLUMN_TYPE	EXTERNAL_TYPE
event	evid	1	ownedid!	i9	9	number(9)	NNSA KB Core	n	-1	primary key	long
event	evname	2	-	a32	32	varchar2(32)	NNSA KB Core	y	-	descriptive data	string
event	prefor	3	orid	i9	9	number(9)	NNSA KB Core	n	-1	unique key	long
event	auth	4	-	a15	15	varchar2(15)	NNSA KB Core	y	-	administrative data	string
event	commid	5	commid	i9	9	number(9)	NNSA KB Core	y	-1	administrative data	long
event	lddate	6	-	a17:YY/MM/DD HH24:MI:SS	17	date	NNSA KB Core	n	-	administrative data	date

2.2 RowGraph

DBUtilLib, the library used by all of the DBTools applications, uses concepts from graph theory [2] to build a directed graph of related Rows in memory.

A graph is a mathematical concept involving a collection of vertices and a collection of edges. Edges are the connections between pairs of vertices. 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. Figure 1 is an illustration of a simple directed graph where A is connected to B and C, B is connected to A, and C is connected to B:

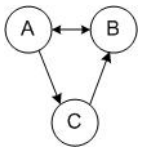

Figure 1. Example of a Simple Directed Graph

An Entity Relationship Diagram (ERD) for tables in a database is an example of a directed graph. The database tables are the vertices and the relationships created by the foreign keys are the bidirectional edges. Consider Figure 2 which illustrates the ERD for the core tables of the NNSA schema [3]. Each box in the diagram represents a table vertex, and each line connecting two tables represents an edge.

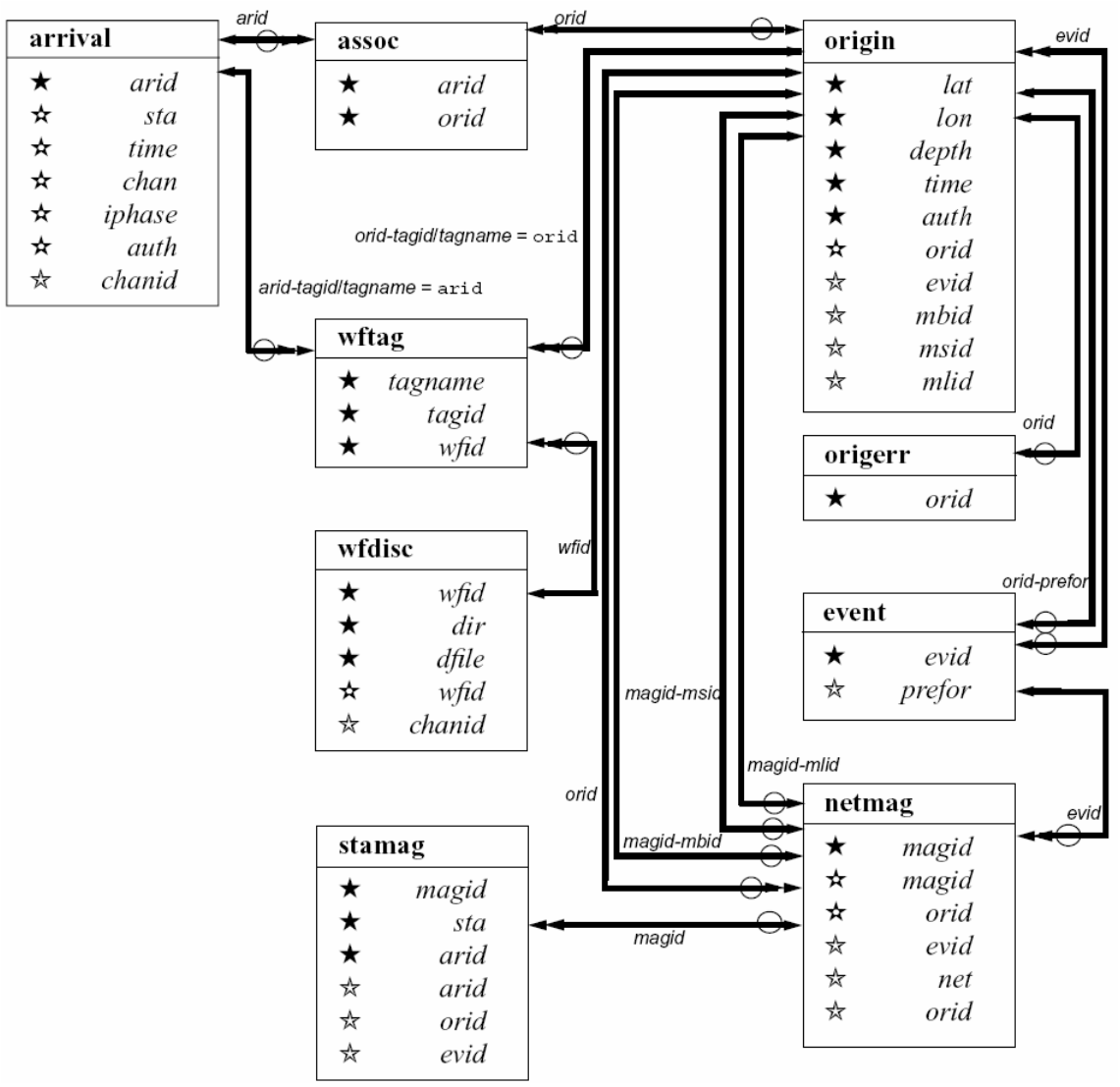

Figure 2. NNSA Core Schema ERD

There can be many edges between two tables, and each edge is labeled with information indicating which foreign key is used to relate the tables. In Figure 2, the **arrival** table is related to the **assoc** table through the *arid* column. This means that if an **arrival** row has an *arid* of 123 and an **assoc** row has an *arid* of 123, those rows are related.

DBUtilLib is able to represent ERDs in memory as directed graphs. The vertices in this graph are the tables in the ERD and the edges are generic SQL select statements that can be used to find related data from the tables the edge connects. (The SQL statements are generic because they do not contain any actual table data.) Consider again the edge labeled *arid* that connects the **arrival** and **assoc** tables. Within DBUtilLib, this edge would be represented by the following SQL:

```
SELECT *
FROM assoc
WHERE arid = #arid#
```

This SQL statement indicates that, given a specific row from an **arrival** table, related rows can be found in the **assoc** table by using the *arid* column.

The #arid# notation is a placeholder for a value will later be substituted into this where clause when looking for an **assoc** row with a particular *arid* value. Thus, given a row from an **arrival** table with an *arid* value of 123, the #arid# in the where clause is replaced with the values 123, and related rows in the **assoc** table can be found by executing the following SQL:

```
SELECT *
FROM assoc
WHERE arid = 123
```

DBUtilLib uses this concept of a directed graph not only to represent an ERD in memory, but also to create RowGraphs. The vertices in this directed graph are no longer tables, but are instead rows (in-memory representations of database rows) from tables represented in the in-memory ERD. While ERDs have bidirectional edges (connectivity between tables can be explored in either direction), RowGraphs can have either unidirectional or bidirectional edges. Having unidirectional edges can be useful if an application is only interested in finding connected rows in one direction but not the other.

A RowGraph is constructed by starting with one or more rows in the database and for each row:

- determining which table that row belongs to
- retrieving the SQL for all of the edges emanating from that table from the in-memory ERD
- replacing column names enclosed in ## in that SQL with the row's value for the column name within ##
- executing the SQL to find related rows

The above steps are repeated until all related rows have been found and added to the RowGraph.

From the application development point of view, a RowGraph is extremely useful. Once a set of database tables has been identified, and the relationships / edges between those tables have been defined, one may specify an initial row in a single table in the database and extract all the other rows (from tables defined in the ERD) related to that row, regardless of how many tables are involved. If the tables and relationships are properly defined, then the application will have in memory only the rows needed for processing. The application will also have the information needed to efficiently traverse this data set and access the rows with which it needs to interact.

2.2.1 RowGraph Example

The easiest way to understand RowGraphs is with an example. Tables 3-7 contain some sample data from **origin**, **event**, **assoc**, **arrival**, and **sitechan** tables. Due to space limitations, not all columns are shown for each table. However, the columns necessary to demonstrate RowGraph functionality are present.

Table 3. Sample Origin Data

LAT	LON	DEPTH	TIME	ORID	EVID	JDATE	GRN	SRN	ETYPE	AUTH
49.9112	78.9267	0	449385550.8	45744	44726	1984089	329	28	en	THUR-BAL

Table 4. Sample Event Data

EVID	PREFOR	AUTH
44726	45744	THUR-BAL

Table 5. Sample Assoc Data

ARID	ORID	STA	PHASE	DELTA	SEAZ	ESAZ
1896847	45744	BRVK	P3KP	6.251	116.82	303.57
1896848	45744	BRVK	Pg	6.251	116.82	303.57
1896849	45744	BRVK	Sn	6.251	116.82	303.57
1896850	45744	BRVK	Lg	6.251	116.82	303.57

Table 6. Sample Arrival Data

STA	TIME	ARID	JDATE	CHANID	CHAN	IPHASE	AUTH
BRVK	449385644.3	1896847	1984089	53	SHZ07	P3KP	LANL
BRVK	449385665.4	1896848	1984089	53	SHZ07	Pg	LANL
BRVK	449385714.0	1896849	1984089	53	SHZ07	Sn	LANL
BRVK	449385745.0	1896850	1984089	53	SHZ07	Lg	LANL

Table 7. Sample Sitechan Data

STA	CHAN	ONDATE	CHANID	OFFDATE	CTYPE	EDEPTH	HANG	VANG	DESCRIP
BRVK	SHZ07	1972102	53	2286324	n	0.015	-1	0	short-period vertical

Given these tables, a simplified representation of the in-memory ERD is shown in Figure 3.

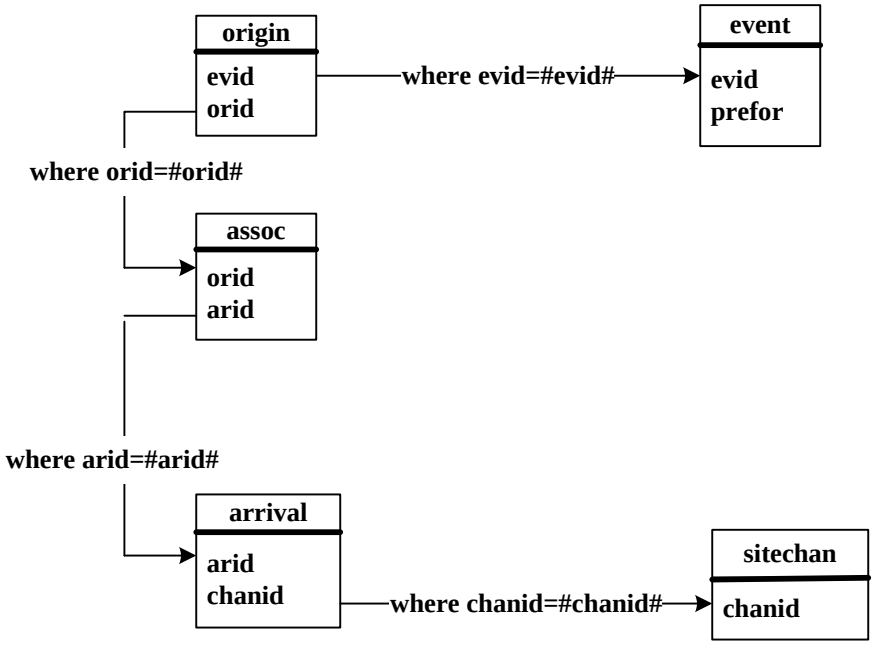

Figure 3. In-Memory ERD for Data from Tables 3-7

This example RowGraph construction will begin with the row of **origin** data. The **origin** table has connections to the **event** table and the **assoc** table. Note that the **origin** table and the **event** table are connected with the

```
where evid = #evid#
```

where clause. To determine if any **event** rows belong in the RowGraph with the **origin** row, the #evid# in the above where clause is replaced with the value from the *evid* values from the **origin** row (44726), and the following SQL is executed:

```
SELECT *
FROM event
WHERE evid = 44726
```

There are no unvisited connections leading from the **event** table to any other tables, so no further processing is necessary for the retrieved **event** row.

Next, the RowGraph construction must determine if there are any **assoc** rows connected to this starting row. Using the

```
where orid=#orid#
```

where clause, the 4 **assoc** rows with the same *orid* as the starting **origin** row (*orid* 45744) are determined to be connected to the starting **origin** row.

The **assoc** table contains connections that have not been explored. Thus, for each of those 4 **assoc** rows connected to the starting **origin** row, SQL select statement must be executed to find connected **arrival** rows. Since there are four different *arid*s, four different SQL statements will be executed:

```
SELECT * FROM arrival WHERE arid = 1896847
SELECT * FROM arrival WHERE arid = 1896848
SELECT * FROM arrival WHERE arid = 1896849
SELECT * FROM arrival WHERE arid = 1896850
```

Four **arrival** rows are retrieved.

Lastly, there is an unexplored connection from the **arrival** table to the **sitechan** table. The RowGraph must determine if there are connected **sitechan** rows that belong in the RowGraph. It is interesting to note here that each **arrival** row has the same *chanid* value. Thus, all four **arrival** rows connect to the same **sitechan** row. However, the RowGraph will only contain one copy of this row in order to avoid duplicate Rows in memory.

Figure 4 contains a rough representation of what this RowGraph looks like in memory:

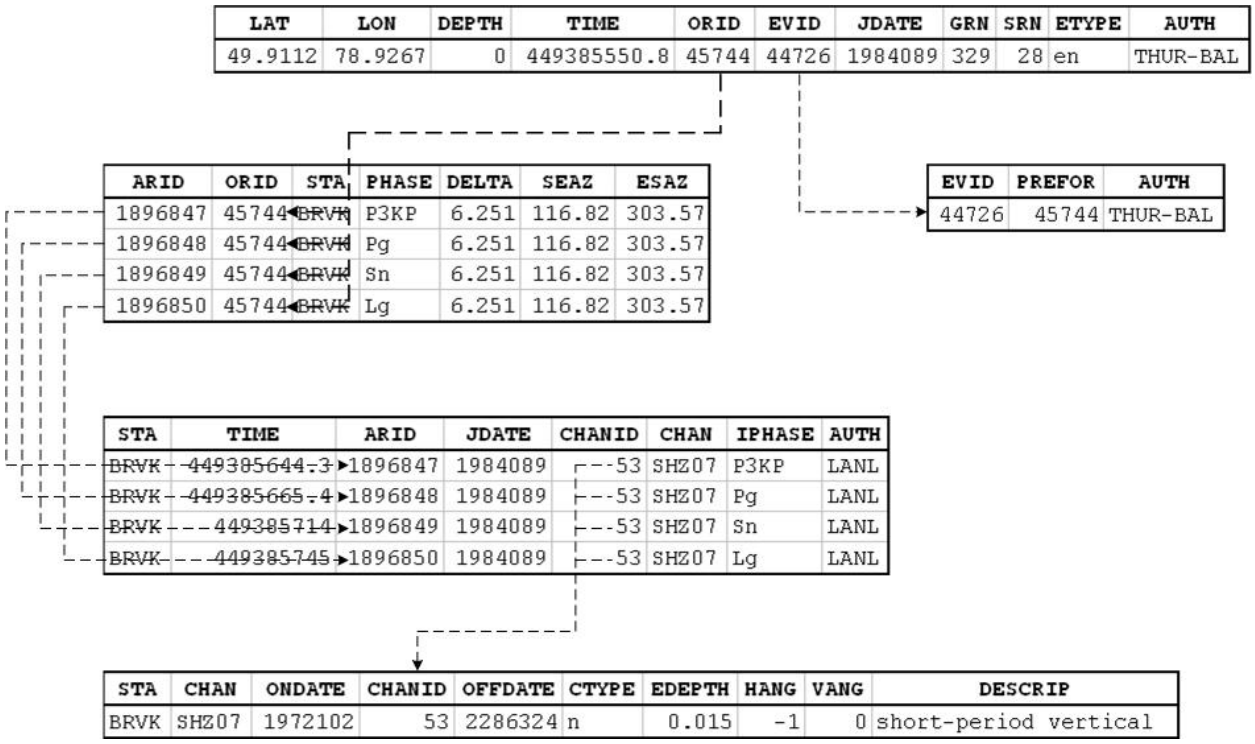

Figure 4. RowGraph for Sample Data in Tables 3-7

2.3 User-Defined Equality

Throughout this document, references are made to "user-defined equality". As the phrase implies, this is when the user specifies what it means for two rows to be equal. Typically, two rows are considered to be equal when every value for every column in those two rows is exactly the same. However, this is not always the desired behavior.

Consider a scenario where a row must be inserted into a database table. Before inserting that row, a check must be made to ensure that that row does not already exist in the database. Checking the table for the existence of a row where every single column's value is identical to the column values of the row to be inserted is highly time consuming. Often, the user does not care if all of the column values are identical, only certain ones.

This is where user-defined equality is useful. By using a SQL where clause, the user defines what rows are equal to the row in consideration (i.e., any rows returned by executing a SQL select statement on the table with the user specified where clause).

As an example, consider an **origin** row that needs to be inserted into an **origin** table. The user has specified the equality for **origin** rows using the following SQL:

```
WHERE lat = #lat# AND lon = #lon# AND depth = #depth# AND
  time = #time# AND auth = #auth#
```

The ## notation indicates places in the SQL statement that need to be populated with values from the row to be inserted. To determine if a row equal to the row to be inserted exists in the table, the following SQL is executed:

```
SELECT *
FROM origin
WHERE lat = #lat# AND lon = #lon# AND depth = #depth# AND
  time = #time# AND auth = #auth#
```

where the ## values are replaced with actual values from the row that is to be inserted. If this SQL select statement returns no rows, then there are no rows in the table that are equal to the row to be inserted. If this SQL select statement returns one or more rows, then one or more rows exist in the target table that are equal to the row to be inserted, and that row should not be inserted.

2.4 Merging

One of the initial goals when developing DBTools was to facilitate merging of data from one data source into another without creating any duplicates, with proper associations maintained in existing data after it has been merged, and with proper management of ID numbers. This is accomplished with the merging capabilities that have been built into DBUtilLib that are used by the DBTools applications and tools.

DBUtilLib's merging functionality implements methods to merge a RowGraph (a set of connected rows), created from data in source tables, into a set of target tables. While the target tables must be of the same type as the source tables, they cannot be the source tables themselves. In other words, you cannot merge a table's data into itself.

Merging information from one set of tables into another presents a number of challenges. The first is that the IDs in the source rows may conflict with the IDs in the target tables. To overcome this, the IDs in the source rows are renumbered using new ID values that do not currently exist in the target tables.

Another challenge when merging rows into a different schema is avoiding the duplication of rows in the target tables. Before inserting a row from the source RowGraph into the target table, the merging code checks to see if a row equal to the row to be inserted exists in the target tables. If such a row is found, Merge will refrain from duplicating the information in the target table by not inserting the source row.

Yet another issue arises when the merging code determines that it cannot insert a row from the source RowGraph – other rows in the RowGraph may be connected to the row that is not being inserted. If that row is not inserted into the target, then the rows that are connected to it will have dangling foreign key pointers once they have been inserted into the target. The merging code avoids this by renumbering IDs in the source RowGraph; IDs in rows that refer to a row that will not be inserted are renumbered to instead refer to the row in the target equal to the row that was not inserted. This preserves the necessary foreign key relationships.

2.5 Undo Capability

Given that the DBTools has the power to modify data within the database, it is essential that what has been done can be undone. DBTools applications have the capability to spool "undo sql" statements to an UndoSQL Table in order to reverse what the application has done. "Undo SQL" is just SQL statements that when executed will restore the database to its former state.

Table 8 contains an example of a few rows from an UndoSQL table that reverses the inserts performed on some tables.

Table 8. Rows from an UndoSQL Table

UNDOID	STATEMENT	LDDATE
1	commit;	8/15/2007 8:52
2	DELETE FROM test_event WHERE LDDATE = TO_DATE('2007-08-15 08:52:29','yyyy-mm-dd hh24:mi:ss');	8/15/2007 8:52
3	DELETE FROM test_origin WHERE LDDATE = TO_DATE('2007-08-15 08:52:29','yyyy-mm-dd hh24:mi:ss');	8/15/2007 8:52
4	DELETE FROM test_origerr WHERE LDDATE = TO_DATE('2007-08-15 08:52:29','yyyy-mm-dd hh24:mi:ss');	8/15/2007 8:52
5	DELETE FROM test_arrival WHERE LDDATE = TO_DATE('2007-08-15 08:52:29','yyyy-mm-dd hh24:mi:ss');	8/15/2007 8:52

To "undo" what a DBTools application has done, simply execute the SQL in the UndoSQL table in reverse order (in Table 8, this would execute the SQL statements in decreasing order from 5 to 1).

3. DBTOOLS APPLICATIONS

This section gives a brief overview of each of the DBTools applications. More detailed information for each application, including how to use each application, is available in the DBTools User's Manual delivered with the DBTools software.

3.1 EvLoader

EvLoader is an application that merges one or more events from a source event table into a target event table. All information that is linked to the source event row(s) is also merged. The user can specify how and what information is related to the source event.

EvLoader operates in two modes. In the first mode, origins in the source event are merged with origins in the target event based on evid number. This mode was developed for users who may create an origin (and associated event) and then pass it through a series of refinement steps wherein the updated information is saved as a set of new origins in different origin tables (but all still linked to the original event). For example, an automatic data processing system might build the original origin (and event), and then an analyst may review and refine that origin and save their work as a new origin (but still the same event). A subsequent analyst might then review and refine that result and save it, and so on. To properly combine the whole series of origins into a single table, one could operate EvLoader in the first mode on each of the origin tables, one by one. Source origins in the source are merged with target origins that have the same evid number as the source origins.

In the second mode, source events are merged with target events based on spatial/temporal correlation. Thus, source origins are merged with target origins that they are close to in space and time. The primary purpose of this mode is to group equivalent origins (i.e., different event hypotheses) from different catalogs. In this spatial/temporal correlation mode, origins that are members of the same event in the source tables will remain members of the same event in the target tables. This is not necessarily true for origins that are members of the same event in the target tables; these events can be broken and the origins reassigned to other events.

In addition to merging event and origin rows, EvLoader also merges in all other data related to those events.

3.2 DTX

DTX (DaTa eXchange) recognizes three information storage formats:

1. a set of database tables
2. a set of ASCII flat files
3. XML

These three formats correspond directly to the three different data access types recognized by DBUtilLib.

DTX can convert information from any of these formats (the "source" format) into any of the others (the "target" format). These formats do not have to conform to the same schema since DTX is also capable of converting data from one schema definition to another.

An example of where the source and target schemas might be different is when DTX is reading data from one schema (e.g. CSS 3.0[1]) and writing it out to a different schema (e.g. NNSA KB Core[2]). Often, the data must be converted from the CSS 3.0 format into the NNSA KB Core format, and DTX is able to do this. There are limitations to the conversions DTX is able to perform. If a data value is in i8 format in the source, it can easily be converted to i9 format in the target since no data is lost. However, if data is in i9 format in the source and must be converted to i8 format in the target, this conversion may not be possible if it would result in the loss of data. If the target schema does not include columns that are in the source schema, then DTX just drops those columns. If the target schema includes columns that the source schema did not, and those columns are allowed to have NA Values, then DTX just adds those columns with NA Values for their values.

Some examples of where DTX is useful:

- Loading flat file data into database tables
- Writing database table data (or a subset of it) into flat files
- Moving data (or a subset of it) from one set of database tables to another set of database tables
- Converting data in a set of database tables from one schema definition to another schema definition

When DTX loads data into the database, it does not simply insert source data into the target - it merges that data. Merging (which is discussed in more detail in Merging section) ensures that the data in the target does not contain duplicate information, that all source data that was referentially connected in the source remains connected in the target, and that all source data is renumbered properly in order to avoid ID conflicts.

3.3 Remap

Given two sets of tables, Remap will generate a remap table which relates source identification numbers to target identification numbers. Source and target identification numbers are "related" when the source and target identification numbers refer to source and target rows which are "equal" where equality is defined by the user. (See the User-Defined Equality section for more information.) Consider the following row of information from a source:

orid	lat	lon	depth	time	auth
123	50	78	0	246942424	someone

[1] Schema version 3.0 for the Center for Seismic Studies [4]
[2] Schema for the core tables for the National Nuclear Security Administration Knowledge Base [3]

and a target:

orid	lat	lon	depth	time	auth
456	50	78	0	246942424	someone

Assume the user has defined "equality" as

```
WHERE lat = #lat# AND lon = #lon# AND depth = #depth# AND
time = #time# AND auth = #auth#
```

Given this definition of equality, these two rows are equal. Thus, a (simplified) remap entry would contain the following information:

id_name	original_id	current_id
orid	123	456

3.4 WFMerge

WFMerge is a waveform merging application. Because waveforms are typically stored on disk, database tables are needed to store the metadata for the waveform files. This application handles the merging of binary waveform files and their associated database table information from a source into a target. WFMerge assumes that the **wfdisc** table contains this waveform metadata.

3.5 Unloader

Unloader "unloads" RowGraphs from the database by deleting one or more specified rows from the database as well as all of the rows that are connected to the specified row(s). Rows are only deleted if doing so will not violate any foreign key relationships in the database or leave any rows referencing a non-existent row. For example, an **arrival** row linked to an **origin** row through an **assoc** relation may also be linked to different **origin** row through another **assoc** relation. Deleting the first **origin** would delete the first **assoc** relation, but not the **arrival** row nor the **assoc** relation to the second **origin** row.

3.6 Parallelyze

Parallelyze is an application that takes parameter files for a DBTools application, splits them into parameter files that are conducive to that application being run in parallel, and then launches parallel versions of that application. This can dramatically improve performance for cases where a large data set is being processed (e.g., EvLoader).

Because the intent of the Parallelyze application is to run applications in parallel, this is only useful if these applications are being run in parallel on a multi-processor machine. Otherwise, the benefits of running in parallel are minimal.

3.7 DBCompare

DBCompare is a tool for comparing tables for equality (see the User-Defined Equality section). Using the user's definition of equality, DBCompare compares the data in all of the tables in the source schema with tables of the same type in the target schema. This tool can be very useful for regression testing of an application that produces a set of database tables as the output for its tests.

3.8 EventTable

EventTable creates a new **event** table using information from an **origin** table. This application is often used prior to running EvLoader when dealing with data sets that do not already include **event** tables.

An event table typically has the following columns:

evid, evname, prefor, auth, commid, lddate.

An **origin** table row has an *evid* and an *auth* that can be used for the *evid* and *auth* columns of an **event** row as well as an *orid* that can be used for the *prefor* in an **event** row. If there are multiple **origin** rows with the same *evid,* the **origin** row that has the *auth* with the highest ranking will be chosen as the source of values for the new **event** row. Author rankings are specified in an author ranking table.

The *evname* and *commid* columns are not populated with information from the **origin** table, but are instead set to their NAValues. The *lddate* is just set to the current date at the time EventTable is run. The resulting **event** table has a number of rows equal to the number of unique *evid*s in the **origin** table used to create the **event** table.

EventTable can also create an **event** table from an **origin** table where all of the *evid*s are set to their NAValues. In this case, one **event** row is created for each **origin** row with *evid*s assigned incrementally starting from 1.

3.9 DatabaseViewer

DatabaseViewer is a tool for viewing sets of related data in the database. Since data is read into memory (and kept there), it is not possible to read in the entire contents of the database. DatabaseViewer is not intended to display all of the data in a database, but instead to facilitate viewing *related* data.

3.10 TableCreator

TableCreator creates tables in the database that conform to a specified schema definition. This is useful since table creation scripts can quickly become obsolete as schema definitions change.

4. SYSTEM REQUIREMENTS

4.1 Java

The DBTools applications and tools are written in Java and compiled to Java bytecode. The system on which these applications are being executed must have a version of Java compatible with the version of Java used during compilation. If a system has Java, it can run DBTools.

At the time of publication of this document, DBTools is compatible with version 1.5.0_07 as well as later versions of Java 1.5.

4.2 Platform

Because DBTools is written entirely in Java, it should be able to run on any system with a compatible version of Java installed.

DBTools has been tested at Sandia National Laboratories on UNIX running Solaris 8 and Solaris 10, Linux, and Windows XP.

4.3 Table Definition Table

All of the DBTools applications and tools require a Table Definition Table to function. DBTools is able to use Table Definition Tables in either database table or flat file formats.

4.4 Data Source (Database tables, Flat files, XML)

The main focus of all of the DBTools applications and tools is interaction with data. Thus, access to a data repository in an acceptable format (e.g., database tables, flat files, or an XML file) is required.

DBTools has been tested using all three data formats. Currently, the only database that DBTools has been tested against is an Oracle 9i database. We are currently beginning the process of testing against Oracle 10g.

4.5 Parameter Files and Environment Settings

Most of the DBTools applications and tools are command line interfaces. The only acceptable way to pass information to the applications and tools is through means of parameters files. A

small subset of the parameters that do not typically change between applications can be set in environment variables, but a parameter file is always required.

4.6 Size

The DBTools distribution (as a .tar.gz file) is roughly 15.5 MB.

5. FUTURE DEVELOPMENT

The DBTools applications and DBUtilLib are mature, so future development is focused on the creation of reliable Graphical User Interfaces (GUIs) for the most commonly used applications.

Along with this GUI development, considerable effort is being made to improve the help system available within these GUIs as well as the User's Manuals for all applications.

This page intentionally left blank.

6. REFERENCES

1. Sun Microsystems, “Java Technology”, August 2007; http://java.sun.com.

2. R. Sedgewick, *Algorithms in Java, 3rd Edition*, Addison-Wesley, 2004, Part 5 – Graph Algorithms.

3. D. Carr, “National Nuclear Security Administration Knowledge Base Core Table Schema Document”, Sandia National Laboratories, 2002, SAND Report SAND2002-3055.

4. J. Anderson, W.E. Farrell, K. Garcia, J. Given and H. Swanger, “Center for Seismic Studies Version 3 Database: Schema Reference Manual”, 1990, Technical Report C90-01.

This page intentionally left blank.

GLOSSARY

Duplicate Row
A duplicate row is one that has the same values in all of the columns being examined. Typically, the user defines which columns should be checked for equality to determine if one row is a duplicate of another. See the section on User-Defined Equality for more information.

Column
A database table contains one or more columns. A column has a particular data type associated with it, and rows with values in that column must adhere to that data type.

Dangling Foreign Key
A foreign key that refers to a value that does not exist.

Equality
See User-Defined Equality.

ERD
Entity Relationship Diagram. A diagram that conveys how database tables are related to one another.

Foreign Key
A column (or set of columns) in a table that refer to a column or set of columns in another table. Foreign keys are used to associate or relate rows from one table to rows in another table.

IDs
In the context of DBTools, IDs are numeric identifiers for a table.

ID-Owner Table
The table who owns ID and whose primary key is that ID. For example, in the NNSA schema, a column named orid occurs in many tables. Thus, orid is an ID. However, only one table owns that ID – the origin table. All other tables with orid IDs in them have orid values that are foreign keys to the origin table.

Merge
DBTools functionality that loads data from source tables into target tables while: a) not duplicating data, b) properly renumbering IDs, and c) maintaining existing relationships between data.

NAValue
A non-null value to use to indicate that no value is available for a given column in a row.

Non ID-Owner Table
A table that does not own an ID.

Owned ID
The ID that an ID-owner table owns.

Primary Key
The primary column (or set of columns) used to uniquely identify a row in a table.

Relationship
Defines how two tables are related.

Remap
A mapping of a source row's ID to a target row's ID when those rows are determined to be equal.

Row
A set of information in the database containing one value for each column.

RowGraph
A set of connected rows (rows that are related).

Schema
A set of tables and information on how those tables are related.

Schema Schema
A schema schema is a set of tables used to describe a schema. These tables have information in about table names, column names, column types, etc – any information needed to be able to construct tables conforming to a particular schema and to verify that data within those tables conforms to that schema.

Source
Information defining the source from which data will be read. This includes the tables, the relationships between those tables, and information on how to access the data (e.g. database connectivity information or the names of flat-files).

Source Schema
Schema for a set of source tables (input).

Source Table
Table in the source – a table from which data will be read.

SQL
Structured Query Language. This is a very powerful language with many different capabilities. In the context of DBTools users, this is the language used when defining queries or relationships between tables.

SQL Select Statement
A SQL statement for extracting data from the database. Typically of the form:
SELECT [* | <*list of column names*>]
FROM *table_name*
WHERE <*restrictions on what data is to be selected*>

Target
Information defining where data will be output / written to. This includes the tables, the relationships between those tables, and information on how to access the data (e.g. database connectivity information or the names of flat-files).

Target Schema
Schema for a set of target tables (output).

Target Table
Table in the target – a table to which data will be written.

Table
A set of data. A table is composed of rows and columns.

Table Definition Table
A collection of metadata about tables that is stored independently of the tables and the code.

Table Type
A definition of a table's structure. This consists of the number of columns, the columns' database types, the columns' Java types, the columns' flat file information, and so on. This information is typically stored in a table definition table.

Unique Key
A column (or set of columns) used to uniquely identify a row in a table.

User-Defined Equality
This is where the user defines what it means for two rows from tables that are of the same table type to be equal. Often, two rows are deemed to be equal if they have the same value in every column. However, with user-defined equality, the user can select which column values are to be examined. If the rows have equal values for just those columns, then they are determined to be equal. See the User-Defined Equality section for more information.

This page intentionally left blank.

DISTRIBUTION

1 Michael L. Begnaud
Los Alamos National Laboratory
MS F659
P.O. Box 1663
Los Alamos, NM 87545

1 Julio C. Aguilar-Chang
Los Alamos National Laboratory
MS F659
P.O. Box 1663
Los Alamos, NM 87545

1 Richard J. Stead
Los Alamos National Laboratories
MS D408
EES-11 P.O. Box 1663
Los Alamo, NM 87545

1 Terri Hauk
Lawrence Livermore National Laboratory
MS L-205
7000 East Ave. POB 808
Livermore, CA 94551-0808

1 Stanley D. Ruppert
Lawrence Livermore National Laboratory
MS L-205
7000 East Ave.
Livermore, CA 94550

1	MS0401	Sanford Ballard	05533
1	MS0401	Glenn T. Barker	05533
1	MS0401	Dorthe B. Carr	05533
1	MS0401	Marcus C. Chang	05533
1	MS0401	David P. Gallegos	05533
1	MS0401	James R. Hipp	05533
1	MS0401	Jake S. Jones	05533
30	MS0401	Jennifer E. Lewis	05533
1	MS0401	Elaine M. Martinez	05533
1	MS0401	Christopher J. Young	05533
1	MS0404	James M. Harris	05736

2	MS9018	Central Technical Files	8944
2	MS0899	Technical Library	9536