A Metadata Driven Approach to Performing Complex Heterogeneous Database Schema Migrations

Robert M. Marks[1] and Roy Sterritt[2]

[1] IBM United Kingdom Limited
Belfast, Northern Ireland
rmarks@uk.ibm.com
[2] School of Computing and Mathematics
Faculty of Engineering
University of Ulster
Northern Ireland
r.sterritt@ulster.ac.uk

Abstract. Enterprise software is evolving at a faster rate than ever before with customer's expecting upgrades to occur regularly. These upgrades not only have complex consequences for legacy software but the database upgrade also. This paper discusses the challenges associated with relational database schema migrations which commonly occur with major upgrade releases of enterprise software. The most prevalent method of performing a schema migration is to execute SQL script files before or after the software upgrade. This approach performs poorly with large or complex database migrations and also requires separate script files for each supported database vendor. A tool was developed for a complex database upgrade of an enterprise product which uses XML in a metadata driven approach. The key advantages include the ability to abstract complexity, provide multi-database vendor support and make the database migration more manageable between software releases. This marks an evolutionary step towards autonomic self-migrations.

Keywords: autonomic, database, upgrades, migrations, metadata

1. Introduction

Enterprise software is evolving at a faster rate than ever before with customer's expecting upgrades to occur regularly. As the software evolution becomes more complex, so too can the database upgrade. This complexity can compromise the software design as developers become reluctant to perform large or complex schema changes between software versions [1].

To highlight this complexity consider a motivating real world scenario: A tool was created to upgrade the enterprise software *IBM® Tivoli® Netcool® Configuration Manager (ITNCM)* [13] from version *6.2 to 6.3*. Up until version

6.2 database changes had been essentially straightforward and consisted of an SQL script file appropriately named *"upgrade.sql"*. This was bundled with the software upgrade and contained all the SQL statements that were necessary to update the new database schema.

However, the 6.3 release had substantial database schema and data changes. An issue that was identified with the 6.2 schema was its primary keys were of type *VARCHAR* and these all had to be changed to be of type *NUMBER*. Changing each database *VARCHAR* field to be of type *NUMBER* was non-trivial task as any foreign key links had to be updated.

The total amount of SQL queries needed to update the ITNCM 6.2 schema was approximately four thousand. To produce these changes manually would have taken too long, and with a tight deadline to meet, a different approach to implement the database migration was required.

In this paper the current research in this field is examined, issues with the current industry approach are discussed along with the typical changes required in a database upgrade. A meta-data approach to performing database migrations is then examined and how it can assist the goal of abstracting the schema migration. The remainder of the paper details the "Cutover Tool", which was created for this work, and which uses a meta-data approach to perform a complex real-world multi-vendor database schema upgrade.

2. Current Research

Several approaches exist for migrating a software system such as Forward Migration Method and the Reverse Migration Method [2]. The Forward Migration Method migrates the database before the software whereas the Reverse Migration Method migrates the software application first and the database migration last. Meier [3] categorizes database migration strategies into three main areas. These are data and code conversion, language transformation and data propagation. This work concentrates on the data conversion and data propagation and is not concerned with language transformation.

The migration of a database can happen at various different levels. These database levels include its contextual schema, internal schemas and external schemas [4]. The database migration could be basic e.g. converting schemas and data restructuring. It can also be more complex such as the horizontal and vertical splitting of table data or computing column data from old data [5].
A customer database can be migrated in a phased manner by creating a new database instance, installing the latest schema and then transporting the data from the old instance to the new. If there are insufficient resources to have two simultaneous databases then the migration can be performed on a single live database.

Maatuk et al classify DB migration into two main techniques: Source-to-Target (ST) and Source-To-Conceptual-To-Target (SCT) [26]. The ST approach translates source to an equivalent target, generally without an ICR

(Intermediate Conceptual Representation) for enrichment, utilizing flat, clustering or nesting techniques. SCT essentially has two stages; reverse engineering where a conceptual scheme is derived from the existing DB (e.g. ERM) then forward engineering that conceptual schema into the target [26]. The SCT approach is presented as being especially necessary if the source DBMS e.g. relational, is structurally different from the target, e.g. Object Oriented. In terms of deriving the ICR; Andersson extracts a conceptual schema by investigating equi-join statements [27]. The approach uses a join condition and the distinct keyword for attribute elimination during key identification, Alhajj developed algorithms for identifying candidate keys to locate FKs in an RDB using data analysis [28]. Chiang et. al. presented a method for extracting an Extended ERM (EERM) from an RDB [29] through derivation and evolution of key-based inclusion dependencies [26].

Currently one of the most common methods is to bundle the upgrade software with one or more script files which contain the SQL statements necessary to update the database schema and data [5, 6].

This basic method gets more cumbersome and unmanageable when the differences in the database schema become more complex and / or the volume of SQL statements are in the thousands [7]. This complexity becomes more compounded if there are variations in the database schemas for different customers e.g. custom software functionality.

Various database migration tools exist such as the open source Migrate4j [8] which performs schema changes using Java code and SwisSQL [9] which has the ability to convert SQL queries between database vendors.

Bernstein [10] remarks major productivity gains can be achieved by utilising model management when manipulating schemas. Yan et al. [11] notes however that tools which manage complex queries for data transformation are still in a primitive state. Curino et al. [12] presents a tool which claims to provide “graceful schema evolution” through the use of Schema Modification Operators (SMO's).

The following table illustrates how the cutover tool compares with other migration tools which support multiple database vendors.

Table 1. Illustration how the Cutover tool (reported in this paper) compares with other database schema migration tools on features.

Operations	Migrate4J [8]	SwisSQL [9]	Cutover Tool
Multi-Vendor DB Support	Yes	Yes	Yes
Basic Schema Changes	Yes	Yes	Yes
Manipulate data in place	No	No	Yes
Column type changes	Yes	Yes	Yes
Update of foreign keys	No	No	Yes
Large object manipulation	No	Yes	Yes
Table Merging and Splitting	No	Yes	Yes
Execute scripts	No	Yes	Yes
Dynamically run Java code	No	No	Yes

Ideally advanced automation of the process is the way forward to cope with the complexity. Autonomic Computing, inspired by the sub-conscious biological self-management, has over the last decade presented the vision to remove the human from the loop to create a self-managing computer-based system [20]. Self-updates, self-migration, self-cut-overs, all should be a part of this initiative.

When the Autonomic vision was first presented, it was done so as a 20-30 year research agenda requiring a revolution. Yet at the opposite end of the scale, as it was an industrial initiative, it also attempted to present an evolutionary path for industry to immediately start to consider steps to create self-management in their legacy and systems under-development.

The Autonomic Capability Maturity Model [21] (Fig. 1) was published to acknowledge that autonomicity cannot happen overnight (indeed Strong-Autonomicity may require "Autonomic-Complete" and dependent on the achievement of AI-Complete, as such the Human-out-of-the-total-loop may be more a motivating inspiration than an actual goal). The ACMM motivates the progression from manual, to managed, to predictive, through adaptive and finally achievement of autonomicity. The database upgrades currently fall between levels 1 to 2. The aim of the work reported here is to progress to level 3.

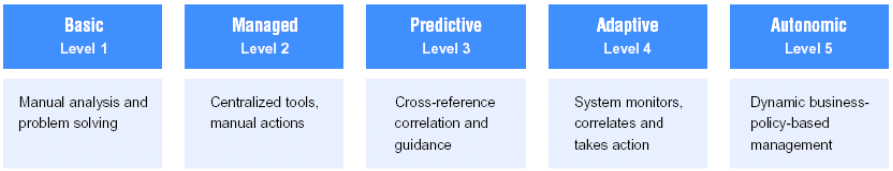

Fig. 1. Autonomic Capability Maturity Model [21]

One of the successes of Autonomic Computing (AC) has been DBMS' due to the implicit management environment nature of RDBMS', together with the self-tuning direction of DBMS research, has aligned with the objectives of AC. Within the SMDB (Self-Managing Database) community itself they have naturally focused on self-optimizing and self-tuning for instance: statistical approaches for ranking database tuning parameters [22], Probabilistic adaptive load balancing for parallel queries [23], but also have looked towards other self-* properties such as self-healing [24].

For DBs and Enterprise Software in general to become fully autonomic, the upgrades must also become self-managing.

3. Issues with Current Approach

The databases that support current enterprise applications have hundreds and even thousands of tables. Maier [14] has observed through empirical analysis that enterprise data models have an average of 536 entity types.

As mentioned in the introduction the most common approach in implementing a database upgrade is to write one or more SQL scripts. This performs well for a diminutive number of simple database schema changes. If however, the schema changes become more complex the migration also becomes error prone and labour intensive.

If multiple database vendors are supported then separate (but conceptually similar) SQL scripts will need to be maintained. It becomes easy for changes to make its way into one script but not another. Another point worth making is that as these scripts become larger they also become more difficult to comprehend as the various changes become lost in a “sea” of SQL.

We have defined a taxonomy of the kinds of change typically required to perform a DB migration. In total we identified eleven kinds of change, which we have subdivided into two categories, "simple" and "complex".
In total there are six "simple" kinds of schema changes. These are as follows:

1. *Add table* - add a new database table.
2. *Delete table* - delete an existing database table.
3. *Rename table* - rename an existing database table.
4. *Add column* - add a database column.
5. *Delete column* - delete a database column.
6. *Rename column* - rename a database column.

These "simple" changes can generally be achieved using a single SQL statement. There are a further five "complex" kinds of change: -

1. *Manipulate data in place* - Updating the existing database content.
2. *Column type changes* - data type migration e.g. changing column type from textual to numeric.
3. *Update of foreign keys* - If a primary key changes then all its foreign keys may require updates.
4. *Large object manipulation* - e.g. changing a BLOB to a CLOB and vice versa.
5. *Table Merging and Splitting* - e.g. one table becomes two or vice versa.

These complex schema changes include anything which may not be performed using a single SQL statement and which may require knowledge of the database schema, such as a list of the foreign key constraints.

In addition to these functional requirements there are several non-functional requirements that affect the migration design. These are as follows: -

1. *Multiple "migrate from" versions* - each software version may have a different schema which could result in an exponential amount of different upgrade scenarios.
2. *Different database vendors* - different migrations are required for each database vendor such as IBM-DB2®, Oracle®, MySQL etc.

3. *Continuous integration* – the migration must be encoded as text so using source control multiple developers can work on and merge their schema changes.

For large database upgrades a declarative metadata based approach proved to be a better solution. The user would define the migration in terms of the six simple and five complex kinds of changes defined above. A tool would then read this metadata and generate the SQL necessary to perform the upgrade.

This approach improves on a simple SQL script as the migration can be expressed in a much more compact form and enables different variations to be easily created. The chances of errors being introduced are reduced as the user is less likely to make a minor SQL error such as an omitted statement. No database specific information is required which means for each upgrade, only a single migration file is required regardless of how many database vendors are supported.

4. A Metadata Approach

A metadata approach would consist of adding a new layer of information which describes the database migration. This layer can be encoded in a variety of ways such as XML [15], JSON [16] and YAML [17] or even plain ASCII text. XML was chosen for this work as it several advantages over ASCII which include the ability to create *user definable structures, hierarchical data, schema validation* and *extensive library support for most programming languages*.

The database XML metadata needs to be read by a piece of software which translates the various lines of XML into SQL statements as illustrated in Fig 2.

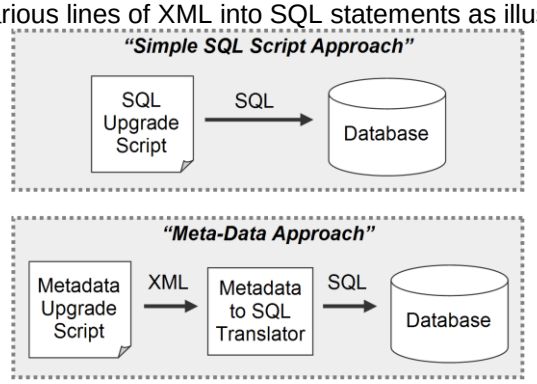

Fig. 2. Standard vs Meta-Data Approach.

For example, the following SQL drops the column *ID* in the *EMPLOYEE* table: -

```
ALTER TABLE EMPLOYEE DROP COLUMN ID;
```

This could be encoded in XML as follows: -

```
<table name=”EMPLOYEE”>
  <column action=”drop” name=”id”/>
</table>
```

The reader may ask why this extra layer of information is required. In the “drop column” example outlined it certainly looks like it is needlessly adding complexity to generate what is a simple SQL statement.

This approach does however bring powerful advantages with more complex tasks, such as changing the type of a foreign key column from a *VARCHAR* to an *NUMBER*.

For example, the following six SQL statements change the *EMPLOYEEID* column of the *SALARY* table from a *VARCHAR* to a *NUMBER*. This column is also a foreign key which points to the ID column in the *EMPLOYEE* table. This is achieved by creating a temporary column named *TEMP_EMPLOYEEID*.

```
ALTER TABLE SALARY ADD TEMP_EMPLOYEEID
VARCHAR2(100);

UPDATE SALARY SET TEMP_EMPLOYEEID = EMPLOYEEID;

UPDATE SALARY SET EMPLOYEEID = NULL;

ALTER TABLE SALARY MODIFY (EMPLOYEEID NUMBER(19));

UPDATE SALARY S1 SET EMPLOYEEID = (SELECT ID FROM
EMPLOYEE WHERE TEMP_ID = T1.TEMP_EMPLOYEEID) WHERE
EMPLOYEEID IS NULL;

ALTER TABLE SALARY DROP COLUMN TEMP_EMPLOYEEID;
```

These six SQL statements could be replaced with the following XML: -

```
<table name=”SALARY”>
  <column action="alter"
          name="EMPLOYEEID"
          type="NUMBER"
          foreign_key="EMPLOYEE.ID"
  />
</table>
```

In the above example, the XML is more concise and intuitive for the user. Also, using this metadata approach context is introduced through the naming and values of the XML elements and attributes.

4.1. Cross Database Support

SQL statements which change a database schema such as ALTER TABLE can vary between different database vendors. The following exemplifies the differences between Oracle, IBM-DB2 and MySQL when altering a column type: -

Oracle

```
ALTER TABLE EMPLOYEE MODIFY (CREATEDBY NUMBER(19));
```

IBM-DB2

```
ALTER TABLE EMPLOYEE ALTER COLUMN CREATEDBY SET DATA
TYPE BIGINT;
```

MySQL

```
ALTER TABLE EMPLOYEE MODIFY CREATEDBY BIGINT;
```

Now the power of using a metadata approach becomes apparent. If you take the scenario of two customers who have the same version of your software, but one customer is on IBM-DB2 and the other is using Oracle and they both require a major schema update.

Using a metadata approach we can have the same XML file which gets translated into the relevant SQL statements for each database vendor. This eliminates the need to have separate SQL script files (each possibly containing several thousand statements) for each database, which must be keep perfectly in sync each time a change occurs on the database schema.

5. The Cutover Tool

A meta-data based database migration tool was developed entirely in Java® and uses the JDOM library [18] for its XML parsing / creation. Its main characteristics include using an XML script to describe the database transformation declaratively. This script is partially generated and can be improved and extended manually. The Cutover Tool then reads the completed script and converts it into SQL statements, which are in turn executed against a target database. Fig 3 illustrates its architecture which is split roughly into three stages: -

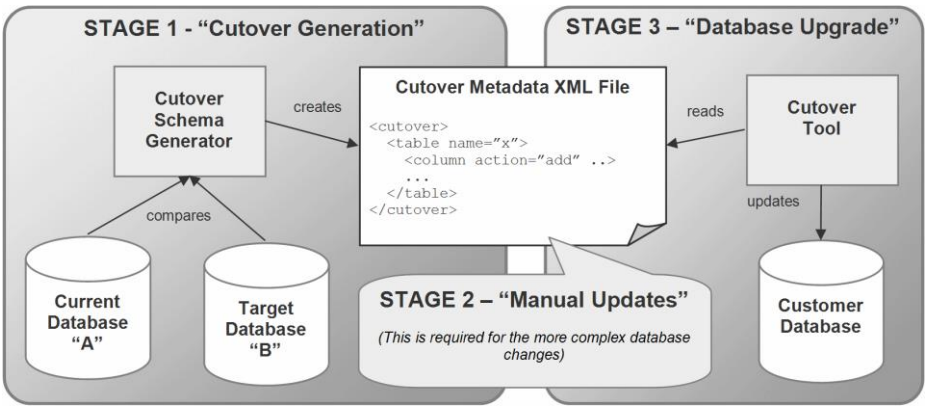

Fig. 3. Architecture of "Cutover Tool"

1. *Cutover Generation* - A smaller tool called the "Cutover Schema Generation" which takes two database connections (the current and target database) and produces a basic cutover XML file specific to the database upgrade.
2. *Manual Updates* - The cutover XML file is manually edited to ensure the generated schema is correct and also to add more complex operations which cannot be generated (e.g. regeneration of primary keys).
3. *Database Upgrade* - Another tool called the "Cutover Tool" takes the edited cutover XML as input and executes it against a customer database as part of the software update.

Each step is now explained in greater detail: -

5.1. Stage 1 - "Cutover Generation"

The first stage involves executing the "Cutover Schema Generator". This tool creates the basic cutover XML file which contains "simple" schema changes as outlined in section 2. It can also partially infer some of the "complex" changes. However, stage 2 is a manual declaration of these.

The generator takes two database connections as its input and compares their tables, column names and column types and writes these differences as *"action"* elements into a XML file.

The order of the database connections is important i.e. database "A" should have the same schema that a customer is currently on, whereas database "B" should be the target database which will work with the target software upgrade. The "action" elements then describe what is necessary to alter the schema of database A to become the schema of database B.

Fig 4 illustrates two basic potential mock databases schemas where “Database A” is the current database schema and “Database B” is the database schema we want to upgrade to.

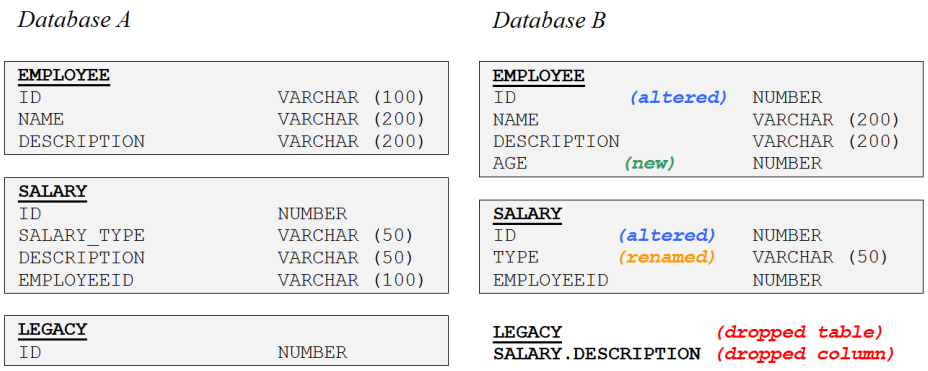

Fig. 4. Two mock database schemas, A and B. The schema differences between database A and B are denoted on the right hand side.

In this example, there are six differences between these two very basic database schemas: -

1. *EMPLOYEE.ID* – column alteration
2. *EMPLOYEE.AGE* – column addition
3. *SALARY.SALARY_TYPE* to *SALARY.TYPE* – column rename
4. *SALARY.DESCRIPTION* – column delete
5. *SALARY.EMPLOYEEID* – column alter
6. *LEGACY* – table drop

The cutover schema generation tool would examine the two databases and by comparing schema data from their respective table's and column's it creates the following cutover XML file:

```
<?xml version=”1.0”?>

<cutover>
  <actions>

    <table name=”EMPLOYEE”>
      <column action=”alter”
              name=”ID”
              type=”NUMBER” />
      <column action=”add”
              name=”AGE”
              type=”NUMBER” />
    </table>

    <table name=”SALARY”>
      <column action=”drop”
```

```
            name="SALARY_TYPE" />
      <column action="add"
            name="TYPE"
            type="VARCHAR(50)"/>
      <column action="drop"
            name="DESCRIPTION" />
      <column action="alter"
            name="EMPLOYEEID"
            type="NUMBER" />
    </table>

    <table name="legacy" action="drop" />

  </actions>
</cutover>
```

The generated XML consists of the main <*cutover*> element, which contains an <*actions*> element which contains three <*table*> elements. Each <*table*> element then contains several <*column*> elements with its *"action"* attribute expressing the type of schema change.

If the generated XML is examined it becomes apparent the column rename was not created successfully. It assumed the *SALARY_TYPE* column was to be dropped and the *TYPE* column was new. This may well be what was required. To guarantee correctness, this stage requires human intervention to ensure the schema changes are correctly specified. The drop and add <*salary*> elements can be removed and replaced with a new *"rename"* <*salary*> action which is correct in this scenario. This ensures the data in the *SALARY_TYPE* column is retained and its column name is all that is modified.

```
<table name="SALARY">
  <column action="rename"
          name="SALARY_TYPE"
          to="TYPE"/>
  ...
</table>
```

When the cutover generation tool was run against the two ITNCM databases it created about 80% of the XML elements required in the upgrade. This equated to 610 "column" elements inside 138 "table" elements which greatly reduced the work load. This figure of 80% is migration dependant and will vary loosely on the ratio of simple to complex updates in each specific database upgrade.

5.2. Stage 2 - “Manual Updates”

The second stage of the cutover involves editing the generated XML file and resolving any discrepancies e.g. column renames instead of column drop and adds.

The other manual updates and additions include the more complex upgrade types detailed in section 2. The implementation of each of these complex types is now discussed in more detail.

1) Manipulate Data in Place

This complex type is concerned with updating the values of the existing data. This is achieved by adding a “value” attribute to the <*column*> element. The value can be one of three kinds as follows: -

a) Arbitrary number e.g. setting column level to 5.

```
<column name=”level” value=”5”/>
```

b) Another column e.g. setting id to employeeid

```
<column name=”id” value=”employeeid”/>
```

c) SQL Statement – where more power is required

```
<column name=”id” value=”SELECT ID FROM EMPLOYEE”/>
```

Where more complex data manipulation is required <*sql*> and <*script*> elements can be used.

2) Column Type Changes

In the previous versions of ITNCM the primary key of all the database tables were of type *VARCHAR*. In version 6.3 it was decided to change these to be of type *NUMBER*. Having a primary key of type *NUMBER* give us several advantages including improved database performance, more efficient storage and the ability to utilise the cross database automatic key generation capabilities of Open JPA (Java based data binding technology) [19].

Some tables in the previous system consisted of special rows where the primary key contained a constant textual value e.g. “Searches”, “Content”. In the new version, special numbers had to be picked which mapped to these constant text strings and the install time SQL content scripts / application code had to be updated accordingly. These special numbers started at -100 e.g.

“Searches” becomes *“-100”*
“Content” becomes *“-101”*

The rational behind the keys starting at -100 was to avoid code which relies on 0 or -1, which the application used at times to denote null, empty or not selected. The decision to update the existing primary keys to minus numbers enabled the values of new primary keys (post cutover) to start incrementally from 1, and therefore not conflict with existing number based data.

To achieve column mapping, the cutover XML file was updated to include the following *<columnmaps>* element which is inserted before the *<actions>* elements. To apply a map to a table column a new optional *"mapid"* attribute has been added to the "column" element. The following example defines a column map called *"users"* which is applied to the *ID* column of the *USERS* table: -

```
<cutover>

  <columnmaps>
    <columnmap name="users">
       <map key="admin" value="-101"/>
    </columnmap>
  </columnmaps>

  <actions>
    <table name="USERS">
      <column action="alter"
              name="ID"
              type="NUMBER"
              mapid="users"/>
    </table>
  </actions>

</cutover>
```

Dynamic mapping is also achievable by utilising special *"sqlkey"* and *"sqlvalue"* elements inside the *<map>* element e.g.

```
  <columnmaps>
    <columnmap name="test_map">
       <!-- SQL key defined -->
       <map sqlkey="SELECT NAME FROM TEST1
                    WHERE ID = 0"
            value="-101"/>

       <!-- SQL value defined -->
       <map key="test"
            sqlvalue="SELECT VALUE FROM TEST2
                      WHERE ID = 1"/>

       <!-- Both SQL key and SQL value -->
       <map sqlkey="SELECT NAME FROM TEST3
                    WHERE ID = 2" />
            sqlvalue="SELECT VALUE FROM TEST3
```

```
                    WHERE ID = 2" />
  </columnmap>
</columnmaps>
```

Defining this data in XML format ensures the mapping can be implemented in various ways depending on the database / environment or even to improve the performance of the upgrade without having to change the underlying XML.

For this work *NUMBER* to *VARCHAR* mapping was used but the *<columnmap>* can manage various mapping scenarios such as *VARCHAR* to *NUMBER*, *VARCHAR* to *TIMESTAMP* etc.

If a primary key column type is altered from a *VARCHAR* to a *NUMBER* we may need some way of regenerating its numbers. If the primary key column has constant values, then these should get mapped first as outlined in the previous section. Sometimes a *VARCHAR* column may contain numbers which are unique, in this situation the regeneration of the field may not need to be required. Regeneration of a column can be specified with the *"regenerate_key"* attribute e.g.

```
<column action="alter"
        name="ID"
        type="NUMBER"
        regenerate_key="true"/>
```

At cutover execution, the column will change from a *VARCHAR* to a *NUMBER* and its values will be regenerated for all existing rows of data (see Fig 5).

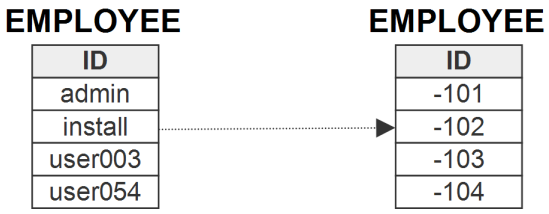

Fig. 5. This example illustrates a database column type change from VARCHAR to INTEGER and value regeneration.

3) Update of Foreign Keys

If a primary key column has its values regenerated (see previous section) and one or more foreign key columns point to the primary key then a mechanism must exist to accommodate this. A column can be specified as a foreign key using the *"foreign_key"* attribute.

For example, the following XML shows the primary key *ID* of the *EMPLOYEE* table being converted to type number and its numeric values being regenerated. It then illustrates the foreign key *EMPLOYEEID* of the

SALARY table should be converted to type *NUMBER* and that it is also a foreign key pointing to the *ID* column.

```
<table name="EMPLOYEE">
  <column action="alter"
          name="ID"
          type="NUMBER"
          regenerate_key="true"/>
</table>

<table name="SALARY">
  <column action="alter"
          name="EMPLOYEEID"
          type="NUMBER"
          foreign_key="EMPLOYEE.ID"/>
</table>
```

The cutover tool will then examine these XML statements and convert them into SQL. It will also ensure that the numeric values of the foreign keys and primary keys are correctly aligned.

4) Large Object Manipulation

A common method of storing large amounts of data in a database involves the use of column types *BLOB (Binary Large OBject)* and *CLOB (Character Large OBject)*. *BLOB's* are used to store binary data such as data from an image file or other proprietary data formats. *CLOB's* are generally used to store large amounts of text. As a database schema evolves between software versions, some fields which were of type *BLOB* may be converted to *CLOB's*. This can be a challenging process and there are various ways to achieve this. One method is to write a SQL function which takes a *BLOB* object and returns a *CLOB* object.

These implementations vary between database vendors but this detail is abstracted away from the XML file. For example, to change a column called *"DOC"* from its existing type *BLOB* to *CLOB* it is very simple:

```
<column action="alter"
        name="DOC"
        type="CLOB"/>
```

This functionality is once again left to the Cutover Tool so that individual database vendors have a different method of converting the *BLOB* to *CLOB*.

5) Table Merging and Splitting

The final complex type is table merging and vertical / horizontal slicing.

Table merging involves taking two tables and combining some or all of their columns and rows of the secondary table into a primary table and then deleting the secondary table if required. Here is the cutover XML which merges the *MANAGER* table into the *EMPLOYEE* table (without delete).

```
<table name="EMPLOYEE"
       merge="MANAGER"
       delete=”no” />
```

Table slicing is the opposite of merging and involves creating a new table from the contents of an old table. Table splitting can be horizontal, which takes rows from a primary table into a new secondary table. Table splitting can also be vertical which moves one or more table columns into a new table.

```
<table name="MANAGER"
       split="EMPLOYEE"
       value="SELECT * FROM EMPLOYEE
              WHERE TYPE = 'manager'"
       delete=”yes” />
```

ORACLE

```
CREATE TABLE MANAGER AS SELECT * FROM EMPLOYEE WHERE
TYPE = 'manager';

DELETE FROM EMPLOYEE WHERE TYPE = 'manager';
```

IBM-DB2

```
CREATE TABLE MANAGER LIKE EMPLOYEE;

INSERT INTO MANAGER SELECT * FROM EMPLOYEE WHERE TYPE =
'manager';

DELETE FROM EMPLOYEE WHERE TYPE = 'manager';
```

As the example show, Oracle can achieve the split in two SQL statements, whereas IBM-DB2 does it in three. This illustrates another example of how the cutover metadata abstracts the detail away by representing the split using a single line of XML.

6) Remaining Issues

An important requirement was that the cutover process should be fully data driven. This ensured a central point of execution for the migration. Other tasks which were required included the ability of the XML file to call SQL scripts. This functionality is useful for loading data into tables and was implemented using the <*script*> element.

```
<script name="sql/insertproperties.sql"/>
```

In the previous example, the cutover tool would read this element and run all the SQL statements that exist in the *"insertproperties.sql"* script file.

Another requirement was the facility to declare SQL statements inside the cutover file XML. This was attended by using the <*sql*> element. e.g.

```
<sql>UPDATE EMPLOYYEES SET ID = 0</sql>
```

The <*sql*> elements can be inserted at the <*table*> level or at <*column*> level depending on its scope within the upgrade.

A final requirement included creating a method of executing compiled Java code from the cutover XML file. This was necessary as some database upgrade tasks were not possible using pure SQL. This could include running complex tasks such as multi-part regular expressions, tree based functions etc. An attribute called functions was then added to the main <*cutover*> element which pointed to a Java class which is loaded at run time using java reflection. e.g.

```
<cutover
  functions="com.ibm.cutover.CutoverFunctions">
```

Individual methods of this class could then be run using the functions element as follows: -

```
<function method="updateUserPreferences"/>
```

At execution the XML would be read and the method executed in a data driven fashion.

5.3. Stage 3 - "Database Upgrade"

Once a user had finished manually editing the cutover XML file the next and final stage was to run the Cutover tool as part of the database upgrade.

The Cutover tool was implemented in Java and uses the JDOM library for parsing the cutover XML file. Execution takes the following two parameters:

1. *Cutover file* - location of the cutover XML file.
2. *Database connection* - location of the database to run the migration against.

After a successful connection to the customer database is established the XML file is parsed in a sequential manner. Fig 6 illustrates a full cutover execution on one table.

Example Execution of the "Cutover Tool"

The following diagram illustrates an example execution of the Cutover Tool. The cutover XML is listed on the left hand side. The tool reads the XML in a sequential manner and the SQL that it generates and executes is listed in the middle column. The mapping between the two is denoted by arrows. This example only contains one table called *"USERS"* but it illustrates both basic and complex operations such as key generation, column mapping, foreign key updates etc.

USERS (Pre-Cutover)

ID	CREATEDBY	EVENT	NAME
admin	install	e0001	Administrator
install	install	e0002	Installer User
user003	admin	e0163	Dave
user054	admin	e0298	Bob

Migrates to →

USERS (Post-Cutover)

ID	CREATEDBY	NAME	LABEL
-101	-102	Administrator	default
-102	-102	Installer User	default
-103	-101	Dave	default
-104	-101	Bob	default

Example Cutover XML File

```
<?xml version="1.0"?>

<cutover>

 <columnmaps>

 <columnmap name="users">
  <map key="admin" value="-101"/>
  <map key="install" value="-102"/>
 </columnmap>

 <columnmap name="license">
  <map key="license" value="-101"/>
 </columnmap>

 </columnmaps>

 <actions>

  <table name="USERS">

   <column action="alter"
           name="ID"
           type="NUMBER"
           mapid="users"
           regenerate_key="true"/>

   <column action="add"
           name="LABEL"
           type="VARCHAR(200)"
           value="default"/>

   <column action="drop"
           name="EVENT"/>

   <column action="alter"
           name="CREATEDBY"
           type="NUMBER"
           foreign_key="USERS.ID"/>

  </table>

 </actions>

</cutover>
```

Generated SQL (Oracle)

```
CREATE TABLE MAPPINGS (NAME  VARCHAR2(255),
                       KEY   VARCHAR2(512),
                       VALUE NUMBER(19));

INSERT INTO MAPPINGS (NAME,    KEY,     VALUE)
              VALUES ('users', 'admin', '-101');

INSERT INTO MAPPINGS (NAME,    KEY,       VALUE)
              VALUES ('users', 'install', '-102');

INSERT INTO MAPPINGS (NAME,      KEY,       VALUE)
              VALUES ('license', 'license', '-101');

ALTER TABLE USERS DROP PRIMARY KEY;

ALTER TABLE USERS ADD TEMP_ID VARCHAR2(100);

UPDATE USERS SET TEMP_ID = ID;

UPDATE USERS SET ID = NULL;

ALTER TABLE USERS MODIFY (ID NUMBER(19));

UPDATE USERS SET ID = (
    SELECT VALUE FROM MAPPINGS
     WHERE KEY = USERS.TEMP_ID AND NAME = 'users')
WHERE TEMP_ID IN (
    SELECT KEY FROM MAPPINGS WHERE NAME = 'users');

SELECT MIN(ID) FROM USERS;

UPDATE USERS SET ID = -102 - ROWNUM WHERE ID IS
NULL;

ALTER TABLE USERS ADD LABEL VARCHAR(200);

UPDATE USERS SET LABEL = 'default';

ALTER TABLE USERS DROP COLUMN EVENT;

ALTER TABLE USERS ADD TEMP_CREATEDBY VARCHAR2(100);

UPDATE USERS SET TEMP_CREATEDBY = CREATEDBY;

UPDATE USERS SET CREATEDBY = NULL;

ALTER TABLE USERS MODIFY (CREATEDBY NUMBER(19));

UPDATE USERS T1 SET CREATEDBY = (
   SELECT ID FROM USERS WHERE TEMP_ID =
T1.TEMP_CREATEDBY) WHERE CREATEDBY IS NULL;

ALTER TABLE USERS DROP COLUMN TEMP_CREATEDBY;

ALTER TABLE GROUPS ADD PRIMARY KEY (ID);
```

Fig. 6. This diagram illustrates a full execution of the Cutover Tool against an imaginary database containing a single table, USERS.

The conversion of XML meta-data to vendor specific database SQL can be compared to the analogy to source code being compiled into vendor specific (e.g. PC, Linux, Mac, etc) classes. Each database vendor has its own syntax but the XML will remain the same. To support a new database the XML to SQL section of the cutover tool can be updated.

We will now discuss in more detail the main items of XML to SQL generation. For example, in the <*mappings*> element, we create and populate a temporary table called *MAPPINGS* from the metadata e.g.

XML

```
<columnmaps>
  <columnmap name="licence">
     <map key="licence" value="-101"/>
  </columnmap>
  <columnmap name="users">
     <map key="admin" value="-101"/>
     <map key="install" value="-102"/>
  </columnmap>
</columnmaps>
```

SQL

```
CREATE TABLE MAPPINGS (NAME VARCHAR2(255),
                       KEY VARCHAR2(512),
                       VALUE NUMBER(19));

INSERT INTO MAPPINGS (NAME, KEY, VALUE)
    VALUES('licence', 'licence', '-101');

INSERT INTO MAPPINGS (NAME, KEY, VALUE)
    VALUES('users', 'admin', '-101');

INSERT INTO MAPPINGS (NAME, KEY, VALUE)
    VALUES('users', 'install', '-102');
```

We could have simply loaded this information into the Cutover Tool application memory but using a temporary database table provides the ability to map database rows using a single SQL as follows.

```
UPDATE USERS SET ID = (
  SELECT VALUE FROM MAPPINGS
  WHERE KEY = USERS.TEMP_ID AND NAME = 'users'
)
WHERE TEMP_ID IN (
  SELECT KEY FROM MAPPINGS
  WHERE NAME = 'users'
);
```

The following XML and SQL illustrate primary key regeneration: -

XML

```
<column action="alter"
        name="ID"
        type="NUMBER"
        mapid="users"
        regenerate_key="true"/>
```

SQL

```
SELECT MIN(ID) FROM USERS;      -- e.g. returns -102

UPDATE USERS SET ID = -102 - ROWNUM WHERE ID IS
NULL;
```

The basic strategy was to pick an arbitrary number e.g. -100 to use as an initial value and then subtract the *ROWNUM* pseudo-column in Oracle (or *ROW_NUMBER* in IBM-DB2) to reset each row. However, if mapping is also performed on the database column (as it is in this example) then mapping occurs before key regeneration. This could result in one or more rows having values and the lowest value of that column must be queried. This value is then used in turn to avoid number conflicts.

Other point to highlight is at the start of each <*table*> element we remove the primary key constraint of the table and re-insert it again after all table alternations have been performed. Temporary columns are also used extensively for the purposes of mapping and populating foreign keys.

A point to make here is that there may exist a more efficient or effective method of implementing the cutover XML to SQL generation. This is perfectly fine and is to be encouraged. The architecture allows for this as different implementations can be created for each new supported database vendor.

Once the database migration was thoroughly tested it was then shipped with the enterprise software upgrade and made available to existing customers.

6. Future Work

The current solution illustrates a successful proof of concept of using meta-data approach to represent a database vendor independent database migration. The tools were implemented as a typical client application for both the cutover generation and migration execution.

To produce the meta-data migration XML using the “Cutover Generator” we assume that the user has two database instances. The first database must be the *“current”* database used by the old software version and the second must be the *“target”* database that the software upgrade will work against. This model forces the creation of the database migration to occur after the software

upgrade has been implemented. However, software engineers generally prefer to develop and test the software / upgrade at the same time and in an incremental and iterative manner.

Work has already begun in moving towards the area of database migrations using an Autonomic Computing paradigm. The basic premise is to create a monitoring agent designed as a client-server / peer-to-peer application which continuously runs in the background for the duration of a software release. The main job of the tool will be to look for changes in the development database and to append these differences into a meta-data file. This incremental cutover file can be constantly validated against a test database using the existing cutover Tool, essentially creating self-migration and self-upgrades functionality into the system. If problems occur user/s can be informed and actions taken appropriately.

To be fully or strong-autonomic, the tool will require to be self-monitoring, self-adjusting and even self-healing which will need considerable research and development in the future.

7. Conclusion

This work presents the problem area of complex database upgrades of enterprise software. Currently the most popular way of executing a database upgrade is to run one or more SQL scripts. This paper examines the various issues associated with this approach. When a database upgrade is complex i.e. requires thousands of SQL statements, different migrate versions and / or support multiple database vendors, then the current SQL script based process can result in an exponential amount of different database migration scenarios. This raises the likelihood of user errors creeping in or scripts becoming out of sync

A taxonomy of the typical changes a migration is comprised of was then defined. This consisted of six “simple” and five “complex” migration tasks. The use of XML meta-data was examined and how it can allow users to express a given migration in a more abstract, simple and concise manner. Using a metadata approach, only a single XML file was required instead of multiple SQL scripts for each database vendor. A cutover tool was created for this work which translates the XML file into the correct SQL statements.

The advantages of this approach also included the ability to run a tool to auto generate most of the “simple” tasks and also some of the more “complex” tasks. This proved to be very useful as it saved substantial effort and increased confidence in the database migration process.

The cutover tool was then bundled into production code and successfully executed against existing large customer databases as part of their software upgrade.

8. Acknowledgement

IBM, Tivoli, Netcool and DB2 are trademarks of International Business Machines Corporation, registered in many jurisdictions worldwide.

Oracle, Java, and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

This paper expands upon a conference publication [25].

References

1. K.-D. Schewe and B. Thalheim, “Component-driven engineering of database applications,” In APCCM’06, volume CRPIT 49, pages 105-114, 2006.
2. M. Brodie, and M. Stonebraker, "Migrating Legacy Systems: Gateways, Interfaces and the Incremental Approach," Morgan Kaufmann Publishers, 1995.
3. A. Meier, “Providing Migration Tools: A Practitioner’s View,” Proceedings of the 21st VLDS Conference, Switzerland, 635-641, 1995.
4. G. H. Sockut and R. P. Goldberg, “Database Reorganization - Principles and Practice”, ACM Computing Surveys, 11(4):371-395, 1979.
5. D. Draheim, M. Horn and I. Schulz, “The Schema Evolution and Data Migration Framework of the Environmental Mass Database IMIS,” SSDBM 2004: 341-344, 2004.
6. M. Elamparithi “Database Migration Tool (DMT) - Accomplishments & Future Directions,” Proceedings of the International Conference on Communication and Computational Intelligence, Kongu Engineering College, Perundurai, Erode, T.N.,India.27 - 29 December,2010.pp.481-485, 2010
7. G. Wikramanayake, W. Gray, N. Fiddian, “Evolving and Migrating Relational Legacy Databases,” 14th Conference of South East Asia Regional Computer Confederation on Sharing IT Achievements for Regional Growth 533-561 Computer Society of Sri Lanka for SEARCC CSSL Sep 5-8, ISBN 955-9155-03-2, 1995.
8. Migrate4j, [Online], http://migrate4j.sourceforge.net, [accessed 7 Nov 2011].
9. SwisSQL Data Migration [Online], http://www.swissql.com, [accessed 7 Nov 2011].
10. P. A. Bernstein, "Applying model management to classical meta data problems." in CIDR, 2003, pp. 209-220, 2003.
11. L. Yan, R. J. Miller, L. M. Haas, and R. Fagin, “Data-Driven Understanding and Refinement of Schema Mappings”, ACM SIGMOD Conference, Santa Barbara, CA, May 2001.
12. Carlo Curino, Hyun Jin Moon, MyungWon Ham, Carlo Zaniolo: The PRISM Workwench: Database Schema Evolution without Tears. ICDE 2009: 1523-1526, 2009.
13. IBM Tivoli Netcool Configuration Manager® (ITNCM), [Online], http://www.ibm.com/software/tivoli/products/netcool-configuration-manager/, [accessed 7 Nov 2011].
14. R. Maier, “Benefits and quality of data modeling - results of an empirical analysis”, LNCS 1157, Springer, Berlin, Cottbus, Germany, Oct. 7-10,1996, pp. 245-260.
15. Extensible Markup Language (XML), [Online], http://www.w3.org/XML, [accessed 7 Nov 2011].
16. JSON, [Online], http://www.json.org, [accessed 7 Nov 2011].

17. YAM, [Online], http://yaml.org, [accessed 7 Nov 2011].
18. JDOM, [Online], http://www.jdom.org, [accessed 7 Nov 2011].
19. OpenJPA, [Online], http://openjpa.apache.org, [accessed 7 Nov].
20. P. Horn's "Autonomic Computing: IBM's Perspective on the State of Information Technology", http://www.research.ibm.com/autonomic/, Oct 2001
21. S. Lightstone, J. Hellerstein, W. Tetzlaff, P. Janson, E. Lassettre, C. Norton, B. Rajaraman, L. Spainhower "Towards Benchmarking Autonomic Computing Maturity",. IEEE Workshop on Autonomic Computing Principles and Architectures (AUCOPA' 2003), Banff AB Canada, Aug. 2003.
22. Biplob K. Debnath, David J. Lilja, Mohamed F. Mokbel, "SARD: A statistical approach for ranking database tuning parameters," Data Engineering Workshops, 22nd International Conference on, pp. 11-18, 2008 IEEE 24th International Conference on Data Engineering Workshop, 2008
23. Daniel M. Yellin, Jorge Buenabad-Chavez, Norman W. Paton, "Probabilistic adaptive load balancing for parallel queries," Data Engineering Workshops, 22nd International Conference on, pp. 19-26, 2008 IEEE 24th International Conference on Data Engineering Workshop, 2008
24. Rimma V. Nehme, "Database, heal thyself," Data Engineering Workshops, 22nd International Conference on, pp. 4-10, 2008 IEEE 24th International Conference on Data Engineering Workshop, 2008
25. Robert M. Marks, "A Metadata Driven Approach to Performing Multi-vendor Database Schema Upgrades," Engineering of Computer-Based Systems, IEEE International Conference on the, pp. 108-116, 2012 IEEE 19th International Conference and Workshops on Engineering of Computer-Based Systems, 2012 doi: 10.1109/ECBS.2012.6
26. Abdelsalam Maatuk, Akhtar Ali, and Nick Rossiter. Relational Database Migration: A Perspective. In Proceedings of the 19th international conference on Database and Expert Systems Applications (DEXA '08), Sourav S. Bhowmick, Josef Kung, and Roland Wagner (Eds.). Springer-Verlag, Berlin, Heidelberg, 676-683. DOI: 10.1007/978-3-540-85654-2_58, 2008.
27. Andersson, M.: Extracting an Entity Relationship Schema from a Relational Database through Reverse Engineering. In: 13th Int. Conf. on the ER Approach, pp.403–419, 1994.
28. Alhajj, R.: Extracting the Extended Entity-Relationship Model from a Legacy Relationals Database. Info. Syst. 28, 597–618, 2003.
29. Chiang, R.H., Barron, T.M., Storey, V.C.: Reverse Engineering of Relational Databases: Extraction of an EER Model from a Relational Database. Data Knowl. Eng. 12, 107–142, 1994.