AUTOMATING SUPPORT FOR REFAC TORIZING SQL DATABASES

Jyrki Nummenmaa, Ari Seppi, Peter Thanisch

University of Tampere, Department of Computer Sciences, 33014 University of Tampere, Tampere, Finland, Jyrki.Nummenmaa@cs.uta.fi, Ari.Seppi@uta.fi, Peter.Thanisch@cs.uta.fi

Abstract. We present a method to automate at least partly the operations needed to refactor an SQL database in the case where a new schema is designed for an existing SQL database that contains data. We base our database refactoring method on intuitive database design operations, such as normalize and denormalize.

Keywords: SQL databases, database maintenance, database refactoring, database evolution.

1 Introduction

Relational database design is a much studied and classical problem. Often the relational database model is based on some higher-level conceptual model. Over time, the concepts and the conceptual level model change and there is a need to update the database. There is solid research on how, for example, the database schema evolution should be designed based on an ER schema.

However, although there is quite a lot of work on schema evolution, we could not find solid research on how an existing relational (or SQL) database should be re-organised, when there is a need to change the database schema. Our conjecture is that this topic has been neglected because researchers were of the opinion that a viable solution to the problem was to redesign the schema at the entity-relationship model level and to migrate the data to the new schema. However, in many production environments, such an upheaval would not be practical. The practical significance of the approach that we propose derives from the fact that schema changes continue to be necessary even after the database has been populated and is in the production environment. It is often essential to preserve as much as possible of the structure of the data (and, thus, information content) existing in the database. It is for this reason that database refactoring cannot simply use conventional entity-relationship modeling tools for this category of design changes.

For modeling the changes in the database, it is necessary to adopt a model for the updates in the database. We choose to model the changes in the database by using operations, which are similar to the typical design operations. Our model is based on using a normalize operation to split a table (schema and data) vertically and a denormalize operation for joining two tables (schemata and data) together. In addition to these, some simpler operations for e.g. changing a single database table are needed. We will also discuss how the operations are mapped into SQL statements.

If the database design produces a trace of atomic changes to the database schema, then it would be possible to just generate a similar sequence of database change operations. However, first of all such a trace is typically not available, as there are uncontrolled and unrecorded changes in the database. Secondly, there may be changes back and forth and the trace does not necessarily exhibit an optimal way to change the database.

Therefore, we assume that we have the old database (schema and data) and the new, desired schema. Our goal is to form such a sequence of operations based on our change operation model that these operations change the database to follow the new schema whilst preserving as much of the information in the database as possible and avoiding unnecessary changes in the data.

There is already a fair number of work on database conversion. The early works on the area concentrate on the Codasyl model, as can be learned from the survey by Sockut and Goldberg [1]. A picture of more recent although not present-day works around the area can be obtained from the survey by Roddick [2]. Presently, the research seems to concentrate in object-oriented systems, such as for example the work by Lerner [3], and reengineering, as for example in the work by Cohen and Feldman [4]. Relational database conversion in general is a widely studied subject, see e.g. the work by Shneiderman and Thomas [5] or the work by Tuovinen and Paakki [6]. Many of the works on relational model, such as the one by Tuovinen and Paakki [6], have their emphasis on maintaining programs, not the actual database. Schema versioning has also been studied – the work by Andany et al. [7] is an example of this line of research. Finally, we bring up the work by Larson et al. [9] where the possibility to identify two attributes to be in fact same in two schemas is studied. The papers listed above are, of course, just examples. However, as for the works where refactoring of so called live databases already containing data is considered, we have only found the book Database Refactoring [8] by Ambler and Sadalege. Ambler and Sadalege propose similar operations of refactoring as we do: the Split Table refactoring.
[8, p. 145] and the Merge Tables refactoring [8, p. 96]. However, we provide a set of operations and an approach on a sound theoretical basis, whilst at the same time remaining practical.

Another work closely related to our research is that Shneideman by Thomas [5], where they propose a set of transformation operations for database conversion. Although we do not adopt their model directly, some of the operations in our model are quite similar or can be seen as taken from their model. However, first of all, Shneideman and Thomas assume that the database is to be kept normalized, which is not always the case in practice. Secondly, they do not provide an algorithm to construct a sequence of their operations to convert a database to obey a given new database schema.

It should be understood that some interaction for the user typically is needed to ensure that the user accepts the suggested conversion and also to ensure that the semantic information carried by naming is correct (ie. the same name means the same thing in different places). However, although this is true, we think that it is possible to give considerable support to the user and even, in fact, propose a sequence of operations to convert an old relational database to conform to a given new schema, by utilizing the semantic information in the dependencies that are known to hold about the data.

Our motivation for this work is practical. Therefore, we base our work on the so-called relational model in the form implemented in SQL databases, and not in the theoretical model [10], as Codd presented it. This means that in particular our model of the dependencies to appear in the database is much more limited than what would be generally available in the relational database theory, see e.g. Atzeni and Antonellis [11].

We have organised our paper as follows. Section 2 presents the SQL-based database model as we use it in this work, and the basic operations we use in database conversion. Section 3 introduces our database conversion algorithm. Section 4 contains some final conclusions.

2 Database operations

We base our database model on a practical choice: the SQL databases. We assume that the reader is familiar with both the basics of relational theory and the basics of SQL databases. We just list out some main differences here shortly. For reviewing relational database theory, we recommend the book by Atzeni de Antonellis [11], and for reviewing SQL databases, we suggest the book by Elmasri and Navathe [12].

First of all, the SQL databases do not generally manage tables as relations, but they allow duplicate rows. Because of this major conceptual mismatch, we will explicitly use the term table in our work, although we use the term attribute and not column name, since the conceptual mismatch is not equally severe in that case. In practice, nearly always a primary key (PK) is defined, therefore eliminating duplicate rows in stored tables.

The management of keys is, at least in practice, different between the relational model and SQL databases. A primary key is typically given a special role. It is possible to specify other keys with the UNIQUE statement. However, the primary key is used in FOREIGN KEY statements. Also, the SQL databases do not manage more general dependencies, like functional or multivalued dependencies. The FOREIGN KEY statements can be used to specify key inclusion dependencies, but more general inclusion dependencies can not be modelled.

As a conclusion, the dependency information is described using PRIMARY KEY, UNIQUE, and FOREIGN KEY statements. We assume that this is also the dependency model in the databases of interest to us. Since the model only includes a primary key, we will use the term prime attribute in table T for an attribute, which belongs to PK(T), and non-prime attribute in table T for an attribute of T, which does not belong to PK(T).

Assume that D is an SQL database, which consists of a set of tables. Each of these tables is described by a tuple <S,R,K>, where S contains the information on the (unique) name and attributes in the table, R is the set of rows in the table, and K is the set of dependencies (one PK, a number of FKs and a number of UNIQUEs). We denote the set of attributes in table T by ATTR(T).

In reality, the database would contain other definitions for things such as users, indices, views, etc., but we only model the part of the database which is our primary focus. Let T=<S,R,K>. Then, we call <S,K> the schema of T. Given a database D, the schema S of database D is the set {<S,K> | <S,K> is a schema of some table in D}.

As a basis for our conversion, we assume that we are given an existing database D with schema S and the desired database schema S'. We further assume that two attributes with the same name in two distinct tables play the same role in both of these tables.

We now go on to define a set of operations, using which we intend to get the database up-to-date. First of all, we have a set of single-relation operations for renaming, adding and removing relations and attributes.
### 2.1 Single table operations

Operations for creating a new table (CREATE TABLE statement), renaming a table (ALTER TABLE statement can be used) or deleting a table (DROP TABLE statement with CASCADE option) are very straightforward and require no further attention here. Of these, of course, only renaming a table preserves the data in the table.

 Renaming an attribute and attribute type conversion are not implemented in all SQL databases – there is variable support for these operations. It may be that several SQL operations are needed to perform these tasks. For type conversion, one may even at worst need to write a simple program with e.g. embedded SQL. However, we skip the simple technicalities here.

 Making changes in the constraints is similarly simple. Using the ALTER TABLE statement one can create new attributes, delete old ones, create and drop constraints (PRIMARY KEYS, UNIQUES, and FOREIGN KEYS). For database conversion, Shneiderman and Thomas [5] particularly introduce the following Promote and Demote operations. Our formalism will not be the same as theirs, but the definitions are equivalent. Let \( T \) be a non-prime attribute in table \( T \). Then we can add \( A \) to PK(\( T \)) with the operation Promote(\( A,T \)). Similarly, let \( A \) be an attribute in PK(\( T \)). Then we make \( A \) a non-prime attribute in \( T \) by the operation Demote(\( A,T \)).

### 2.2 Partitioning and joining tables with equal keys

In some database conversions, one either wants to partition a table into two tables both having the same key or to join together two tables which have the same primary key. Shneiderman and Thomas [5] consider the following Compose and Decompose operations. Again, our formalism differs from that of Shneiderman and Thomas, but the operations are equivalent.

Assume \( T \) is a table and \( \{X_1,\ldots,X_k\} \) is a partition of its non-prime attributes. Then, we can decompose \( T \) into tables \( T_1,\ldots,T_k \) with respective attributes PK(\( T \) \( \cup \) \( X_1 \), \ldots \( \cup \) \( X_k \)) with the operation Decompose(\( T,T_1,\ldots,T_k \)). Then, we can join the tables \( T_1,\ldots,T_k \) using the operation Compose(\( T_1,\ldots,T_k,T \)), which in effects performs a join on \( K \). In SQL, let’s look at how we can construct the data by applying a sequence of queries of the following type:

\[
\text{CREATE TABLE } T_i \\
\text{ AS SELECT DISTINCT } K, X_i \\
\text{ FROM } T
\]

Similarly, assume that \( T_1,\ldots,T_k \) are such tables that they all have the same primary key \( K \) and that the non-prime attributes of \( T_i \) are \( X_i \), and that the sets \( X_i \) are disjoint. Then, we can join the tables \( T_1,\ldots,T_k \) using the operation Decompose(\( T_1,\ldots,T_k \)).

This will get all data from tables \( T_1 \) and \( T_2 \), although null values are likely to be introduced. Now it is possible to use this table (or result) to join in similarly \( T_3,\ldots,T_k \).

As Shneiderman and Thomas point out, Compose is not information preserving in the sense that the source relations lose their separate identity, and they also propose other operations, which overcome this complication [5]. However, since we are only interested in conversion in one direction, we are not going to discuss these other operations here.

It should be noted that under the assumptions given, the Compose and Decompose operations preserve the data in the tables, although Compose loses the identity information of the original input tables.

### 2.3 Denormalize

The above changes in the database are standard and directly available as SQL operations. However, the main idea in our database transformations is to use, whenever possible, intuitive Normalize and Denormalize operations corresponding to normalization of a single table and denormalization by joining two tables together. Although the Denormalize operation should contain no surprises, we describe it in more detail. For this, we will use SQL operations.

In the Denormalize operation first a new table is created and after that the old tables are removed. Let \( T_1 \) and \( T_2 \) be database tables, and assume that \( K \) is the primary key of \( T_1 \) and \( K \) appears as a foreign key in \( T_2 \). Let \( T \) be the new table to be created. For denormalizing \( T_1 \) and \( T_2 \) into \( T \), we define the operation Denormalize(\( T_1,T_2,T \)) with the following operation sequence.
1. Create the new table with data joined from both tables. Note that in some systems it is possible to use a separate SQL statement JOIN.

```sql
CREATE TABLE T AS SELECT Attr(T1), Attr(T2) FROM T1, T2 WHERE T1.K = T2.K
```

2. After the joined table is created the primary key constraint is added. We get a primary key as the union of PK(T1) and PK(T2). Notice that if PK(T1) is a subset of PK(T2), then PK(T)=PK(T2).

3. The unique constraints are created as follows. Let \( Y_1 \) be set of attributes for which there is a unique constraint in \( T_1 \) and let \( Y_2 \) be set of attributes for which there is a unique constraint in \( T_2 \). Then it follows that the union \( Y_1 \cup Y_2 \) is unique in \( T \). PK(T)=PK(T2), then \( Y_2 \) alone is unique in \( T \).

4. If PK(T) appears in some table and that table has an existing foreign key statement to \( T_2 \), it can be changed to point to PK(T). The foreign key constraints in \( T_1 \) and \( T_2 \), are to be inserted for \( T \).

5. Finally the old tables are dropped.

```sql
DROP TABLE T1 CASCADE
DROP TABLE T2 CASCADE
```

It should be intuitively clear that Denormalize preserves the data in the database, if the assumptions given for it hold.

**Example 1.** Suppose we have a database \( D_1 \) with table \( DEPT \) for departments, table \( EMP \) for employees, table \( EMP_PHONE \) for employee phone numbers (separately, because not everyone has a phone number), and a table \( TITLE \) for storing information on the titles of employees (also separately, as not everyone has a title).

Suppose that the tables have the following attributes and primary keys:

- \( Attr(DEPT) = \{D_NO, D_NAME, D_ADDRESS\} \)
- \( PK(DEPT) = \{D_NO\} \)
- \( Attr(EMP) = \{D_NO, E_NO, E_USERNAME, E_NAME, QUALIFICATION, OFFICE\} \)
- \( PK(EMP) = \{D_NO,E_NO\} \)
- \( Attr(EMP_PHONE) = \{D_NO, E_NO, PHONE_NO\} \)
- \( PK(EMP_PHONE) = \{D_NO,E_NO\} \)
- \( Attr(TITLE) = \{D_NO, E_NO, TITLE\} \)
- \( PK(TITLE) = \{D_NO,E_NO\} \)

Assume that \( D_NAME \) and \( D_ADDRESS \) are both unique in \( DEPT \), \( E_USERNAME \) is unique in \( EMP \) and \( D_NO \) in \( DEPT \) appears as a foreign key in \( EMP \) and \( \{D_NO,E_NO\} \) in \( EMP \) appears as a foreign key in \( TITLE \) and \( EMP_PHONE \).

Suppose, further, that we are doing Denormalize\( (DEPT,EMP,NEW_EMP) \). Then we get a new table \( NEW_EMP \) such that

- \( Attr(NEW_EMP) = \{D_NO, D_NAME, D_ADDRESS, E_NO, E_USERNAME, E_NAME\} \)
- \( PK(NEW_EMP) = \{D_NO,E_NO\} \)

and \( E_USERNAME \) is unique in \( NEW_EMP \), because \( PK(NEW_EMP)=PK(EMP) \). Foreign key constraints are to be updated accordingly.

### 2.4 Normalize

The idea of the Normalize operation is to normalize (split) a relation into two. Similarly as with Denormalize, we will describe Normalize in more detail.

Let \( T \) be a table to be normalized so that a new table \( T' \) is formed with \( X = Attr(T') \subseteq Attr(T) \) and \( K = PK(T') \) will be a foreign key in \( T \) and the attributes \( X-K \) (assumed to be non-prime in \( T \)) are to be removed from \( T \). In more detail, we define Normalize\( (T,T',X,K) \) as follows:
1. Create table $T'$:

   \[
   \text{CREATE TABLE } T' \text{ AS SELECT DISTINCT } X \text{ FROM } T
   \]

2. Add primary key $K$ for $T'$.

3. Each attribute $A$ in $X-K$ are to be removed from $T$. Unique and foreign key constraints including attributes in $X-K$ will be removed at the same time from $T$:

   \[
   \text{ALTER TABLE } T \text{ DROP COLUMN } A \text{ CASCADE}
   \]

4. If initially there were unique or foreign key constraints in $T$ containing only attributes in $X$ (\(\text{ATTR}(T)\setminus(X-K)\), respectively), then similar unique statements can be created for $T'$ ($T$, respectively). Notice that it is possible that also other unique constraints hold in the new table.

5. If desired, remove possible duplicate rows in $T$.

6. The key $K$ in $T'$ is made a foreign key in $T$.

   It should be fairly easy to see that if the conditions for the operations are met, the operation preserves the data in the database.

**Example 2.** Considering Example 1 and the table $\text{NEW\_EMP}$ created with the Denormalize operation.

Now we have $\text{NEW\_EMP}(\text{D\_NO, D\_NAME, D\_ADDRESS, E\_NO, E\_USERNAME, E\_NAME})$.

$\text{PK(NEW\_EMP)} = \{\text{D\_NO, E\_NO}\}$. \{\text{D\_NAME, E\_USERNAME}\} is unique in $\text{NEW\_EMP}$, and \{\text{D\_ADDRESS, E\_USERNAME}\} is unique in $\text{NEW\_EMP}$.

The operation $\text{Normalize(NEW\_EMP, DEPT, \{\text{D\_NO, D\_NAME, D\_ADDRESS}\}, \{\text{D\_NO}\})}$ will now give us the initial tables and constraints of Example 1, with the exception that $\text{NEW\_EMP}$ will still have the name $\text{NEW\_EMP}$ instead of $\text{EMP}$, and that we can not automatically infer that $\text{D\_NAME}$ and $\text{E\_ADDRESS}$ are unique in $\text{DEPT}$.

### 3 Database conversion algorithm

Database designers frequently give careful consideration to their choices about which attributes to use as a key. Such considerations might include some special insight into the nature of the business and the pattern of usage of the database. In view of this, we first modify the old database so that it will have a key structure that is identical to the new desired database schema.

Some of the operations introduced in the previous section change the key structure of the database. First of all, we may try to fine-tune tables by promoting non-prime attributes and demoting prime attributes. Secondly, we may normalize and denormalize tables. Thirdly, we may delete tables and create new tables with appropriate primary keys. Apparently deleting old tables and creating new ones will alone be enough to correct the mismatch, but then no data will be preserved, so it is also the last resort to fix the key structure.

Promoting and demoting attributes are strong operations and they should not be done automatically. In the interface that we have designed, the user can specify whether the proposed promote/demote is acceptable.

Let $S$ and $S'$ be two database schemas. We define $\text{NONMATCH}(S,S') = \{T \mid T$ is such a table schema in $S$ that no table schema in $S'$ has the same primary key as $T\}$.

Further, we say that we have unified the key structures of $S$ and $S'$, when both $\text{NONMATCH}(S,S')$ and $\text{NONMATCH}(S',S)$ are empty.

**Algorithm 1.** Key structure unification.

Input: Database $D$ with schema $S$, and the desired new database schema $S'$.

Output: The algorithm will modify $D$ so that the key structures of $S$ and $S'$ are unified. Notice that in each step the modifications are performed into $D$ and $S$ changes, and so may $\text{NONMATCH}(S,S')$ and $\text{NONMATCH}(S',S)$.

1. For each table $T$ in $\text{NONMATCH}(S,S')$, if there is a table $T'$ in $D'$ such that it is acceptable to promote or demote attributes in $T$ so that $\text{PK}(T) = \text{PK}(T')$, promote or demote attributes accordingly.

2. Normalize, if it can be used to reduce the number of elements in $\text{NONMATCH}(S',S)$ without creating new elements in it.

3. Denormalize, if it can be used to reduce the number of elements in either $\text{NONMATCH}(S,S')$ or $\text{NONMATCH}(S',S)$ without creating new elements in them.

4. Drop and create tables with required keys until both $\text{NONMATCH}(S,S')$ and $\text{NONMATCH}(S',S)$ are empty. When creating tables, make the non-prime attributes match the desired table schema in $S'$.

When we say “acceptable” above, it means that a user accepts the operation.
Lemma 1. Algorithm 1 terminates.

Proof. Each step in Algorithm 1 will terminate, because of the way they reduce the number of tables in \( \text{NONMATCH}(S',S) \) or \( \text{NONMATCH}(S,S') \).

Lemma 2. Algorithm 1 unifies the key structure of \( S \) and \( S' \).

Proof. The lemma follows trivially from Step 4 of the algorithm.

Example 3. Let us consider giving a database with schema \( S_1 \) of Example 1 as an input to Algorithm 1. Assume, that the target schema \( S_2 \) has the following tables and primary keys:

\[
\begin{align*}
\text{ATTR}(\text{EMP'}) &= \{\text{D_NO, E_NO, E_USERNAME, E_NAME, TITLE, D_NAME, D_ADDRESS}\} \\
\text{PK}(\text{EMP'}) &= \{\text{D_NO, E_NO}\} \\
\text{ATTR}(\text{EMP_PHONE'}) &= \{\text{D_NO, E_NO, PHONE_NO}\} \\
\text{PK}(\text{EMP_PHONE'}) &= \{\text{D_NO, E_NO, PHONE_NO}\} \\
\text{ATTR}(\text{OFFICE'}) &= \{\text{D_NO, E_NO, OFFICE}\} \\
\text{PK}(\text{OFFICE'}) &= \{\text{D_NO, E_NO}\} \\
\text{ATTR}(\text{QUALIFICATION'}) &= \{\text{D_NO, E_NO, QUALIFICATION}\} \\
\text{PK}(\text{QUALIFICATION'}) &= \{\text{D_NO, E_NO, QUALIFICATION}\}
\end{align*}
\]

Intuitively, the desired target \( S_2 \) may have the following explanation. As the department data hardly ever changes, the department and employee information has been put into a common table to avoid joins in queries. The employee titles are stored in the employee table \( \text{EMP'} \), because it was not reasonable to keep them separated, even though this may mean some NULL values. Some employees have more than qualification, so a separate table was required. Some employees also have more than one phone, so the primary key for the \( \text{EMP_PHONE'} \) includes the phone number. Since very few employees are located in an office, it was chosen to put this information in \( \text{OFFICE'} \) to avoid a large number of NULL values.

Now, the primary keys in \( S_1 \) are as follows:

\[
\begin{align*}
\text{PK}(\text{DEPT}) &= \{\text{D_NO}\} \\
\text{PK}(\text{EMP}) &= \{\text{D_NO, E_NO}\} \\
\text{PK}(\text{EMP_PHONE}) &= \{\text{D_NO, E_NO}\} \\
\text{PK}(\text{TITLE}) &= \{\text{D_NO, E_NO}\}
\end{align*}
\]

and the primary keys in \( S_2 \) are as follows:

\[
\begin{align*}
\text{PK}(\text{EMP'}) &= \{\text{D_NO, E_NO}\} \\
\text{PK}(\text{EMP_PHONE'}) &= \{\text{D_NO, E_NO, PHONE_NO}\} \\
\text{PK}(\text{OFFICE'}) &= \{\text{D_NO, E_NO}\} \\
\text{PK}(\text{QUALIFICATION'}) &= \{\text{D_NO, E_NO, QUALIFICATION}\}
\end{align*}
\]

Therefore, \( \text{NONMATCH}(S_1, S_2) \) comprises of the schema of the table \( \text{DEPT} \). Respectively, \( \text{NONMATCH}(S_2, S_1) \) comprises the schemas of tables \( \text{EMP_PHONE'} \) and \( \text{QUALIFICATION'} \).

With the above intuitive explanation in mind, the only table, where attribute promotion is acceptable, is table \( \text{EMP_PHONE} \). There is no table, where attribute demotion would be acceptable. This way, Step 1 removes the schema of \( \text{EMP_PHONE'} \) from \( \text{NONMATCH}(S_2, S_1) \), but does not change \( \text{NONMATCH}(S_1, S_2) \).

In Step 2, it is possible to normalize \( \text{EMP} \) to obtain \( \text{QUALIFICATION'} \) and modify \( \text{EMP} \) accordingly. This will remove the schema of \( \text{QUALIFICATION'} \) from \( \text{NONMATCH}(S_2, S_1) \) thereby making it empty, without changing \( \text{NONMATCH}(S_1, S_2) \).

In Step 3, it is possible to denormalize \( \text{EMP} \) and \( \text{DEPT} \) into, say, \( \text{EMP''} \) similarly as in Example 1, thus removing \( \text{DEPT} \) from \( \text{NONMATCH}(S_1, S_2) \) and making it empty.

Since after Step 3 both \( \text{NONMATCH}(S_1, S_2) \) and \( \text{NONMATCH}(S_2, S_1) \) are empty, there is no need to drop or create tables in Step 4. This is also intuitively understandable, since no radically new data is introduced or deleted in the target schema \( S_2 \) when compared to the original schema \( S_1 \).

Notably, normalize and denormalize can be used to automatically create UNIQUE and FOREIGN KEY constraints. If, however, the target database schema contains these, then the automatically generated constraints can be used to check for the validity of normalize and denormalize operations.

Assume that the database \( D \) is modified with Algorithm 1 to unify the key structure. When the key structure is unified, then of course the database schemata may still not be equal. However, using the \( \text{Compose} \) and \( \text{Decompose} \) operations, we may further unify the schemata. After that, the necessary final fixes may be done
by type conversion and simply creating new attributes and deleting old ones so that the database schemata become equal. So, after Algorithm 1, the following clean-up steps are needed:

1. Use \texttt{Compose} and \texttt{Decompose} to adjust the attribute structure in tables of \( S \) to match that of \( S' \).
2. Do type conversion, add attributes, and delete attributes so that \( S \) becomes identical to \( S' \).
3. Change the unique and foreign key constraints to fit the new schema.

These clean-up steps will clearly make \( S \) equal to \( S' \). To continue Example 3, in Step 5 we would use Decompose to split \( \text{EMP}'' \) into \( \text{EMP}' \) into \( \text{OFFICE}' \) after which our new database would have the required key and attribute structure.

Notice that by using \texttt{Demote} and \texttt{Compose} we achieve partly the same effect as with \texttt{Denormalize}, but for the management of constraints. Generally, the management of constraints is largely omitted here due to space limitations.

4 Conclusions

Our motivation for this work is largely practical, and therefore we have also implemented our method as a part of a larger collection of database-oriented software development tools. The platform for our implementation is the Fujaba [13] toolset. The core of the Fujaba toolset is a set of UML-based functionalities and the ability to do round-trip engineering from UML to Java and from Java to UML. However, the Fujaba toolset has a plugin architecture, which allows developers to write their own functionalities and add them to Fujaba. We have implemented a database plugin [14] to allow Fujaba users to create an object-oriented interface to their databases. The database refactoring operations can be used through that interface.

References

ENHANCING CONNECTION BETWEEN ONTOLOGIES AND DATABASES WITH OWL 2 CONCEPTS AND SPARQL

Ernestas Vysniauskas, Lina Nemuraite, Algirdas Sukys, Bronius Paradauskas

Kaunas University of Technology, Department of Information Systems, Studentu 50-308a, Kaunas, Lithuania, Ernestas.Vysniauskas@stud.ktu.lt, Lina.Nemuraite@ktu.lt, Sukys.Algirdas@gmail.com, BoniusParadauskas@ktu.lt

Abstract. The goal of the paper is to present the enhanced database schema for storing ontologies considering new features of OWL 2 and possibilities of querying these ontologies using SPARQL. The growing size of ontologies and the scope of their applications require the effective means for storing ontology data that relational databases already have approved. Many existing ontology reasoning tools are using relational databases for this purpose. However, in practice almost all of them are using the straightforward approach restricted to representing instances whereas the effectiveness of processing ontological data may be considerably improved by keeping information about ontology classes, object properties and more advanced concepts in database tables. Previously we have presented the method and tool for transforming OWL ontologies to relational database. Currently, we have extended our representation with novel concepts of OWL 2, the recent Recommendation of W3C. Also, we present a prototype of a tool for extracting ontologies from relational databases and thus allowing the step-wise processing of SPARQL queries where SPARQL is used for querying ontology structures in a main memory and SQL is used for querying instances in the database.

Keywords. Ontology, relational database, OWL 2, SPARQL, SQL, mapping, transformation.

1 Introduction

Ontology descriptions, in particular Web Ontology Language OWL, become more and more widely used in the World Wide Web and other fields as common information systems, data integration and software engineering. Currently, many areas are becoming knowledge-based [12]. Ontologies allow creating of better information systems by empowering them with advanced possibilities for operation and interoperability with other systems by opening access to existing heterogeneous and distributed databases and other information resources. Since 2004, a practical experience with OWL 1 has shown that it lacks several constructs that are often necessary for modelling complex domains [11]. The new developments of OWL 1, initially informally undertaken by some group of its users, in 2009 were issued as OWL 2 – a new W3C Recommendation [10, 23], augmenting the previous OWL with advanced features: qualified cardinality restrictions; complex sub-property axioms between a property and a property chain; local reflexivity restrictions; disjoint, reflexive, irreflexive, symmetric, and anti-symmetric properties; negative property assertions; vocabulary sharing (punning) between individuals, classes, and properties; the richer set of datatypes and datatype restrictions etc.

In our previous work [29, 30] we have proposed the mapping between OWL ontology and relational database and a tool for transforming ontologies into databases. Such an approach is needed because ontology based systems are growing in scope and storages of ontology reasoners are becoming unsuitable. While there are other solutions and tools for keeping ontologies in databases, the most of them are storing only RDF data. Lee and Goodwin, who have proposed the database-centric approach to ontology management support, notice that such methodology is still in its infancy [20]. In our approach, some concepts, e.g. ontology classes and properties are mapped to relational tables, relations and attributes, other (constraints) are stored like metadata in special tables. Using both direct mapping and metadata, it is possible to obtain appropriate relational structures and do not lose the ontological data. In connection with new features of OWL 2 and its supporting tools as Protégé [18] and Pellet [27], we have revised the previous representation and supplemented it with new concepts.

Our approach is well-suited for creating new databases from ontologies, however, in practice ontologies often are used for accessing already existing databases that usually are heterogeneous and distributed. Therefore, methodologies are even more needed for extracting ontologies from existing databases. It is a hard task to automatically obtain meaningful knowledge from such legacy systems without human intervention, so it is worth to ready beforehand ontology structures and to store them in databases for ontology management purposes. In cases when ontologies are being created from databases our approach fits as well, because the database schema obtained by our transformation is capable of the lossless representation of ontological

---

1 The research is pursued according the project proposal “Methodology and Technology Foundations for Semantically-Based Information System Design (SEMIS)”

2 We have in mind the lossless transformation of OWL ontologies formulated using sufficient subset of its concepts because criteria of completeness and performance require for some compromise. Complete representation of OWL in a database is even undesirable as inference in OWL FULL is undecidable [14].
information in databases and the lossless retrieval of this information from databases into ontology reasoning tools. We present a prototype of a tool for extracting ontologies from relational databases, satisfying our schema, and allowing the step-wise processing of SPARQL queries where SPARQL is used for querying ontology structures in a main memory and SQL is used for querying instances in the database.

The rest of the paper is organized as follows. Section 2 presents related works. Section 3 is devoted to mapping of OWL 2 concepts to RDB concepts. Section 4 presents our approach to querying OWL 2 ontologies stored in relational databases. Section 5 draws conclusions and outlines the future work.

2 Related Work

OWL is different from conceptual modelling languages as ER or UML class diagrams as it has richer capabilities to describe classes and to handle incomplete knowledge. OWL 2, the emerging new version of OWL, is more expressive and still allows for complete and decidable computing [11]. Significant improvement in ontology management and reasoning tools has been achieved due to enhancement and additional functionality provided in OWL 2. For example, the widely used Protégé and Pellet systems and graphical OWL notation were extended with additional constructs of OWL 2 [18, 27, 16]. Consequently, we are aiming for the extending and improving our previous OWL2RDB transformation in accordance with new possibilities of OWL 2.

The new features of OWL 2 aim at increasing the relational expressivity of OWL 1 by allowing propagation of constraints along properties: transitivity of properties, subproperty and property chain axioms [10, 11, 23]. Object property axioms now can define reflexivity and symmetry, and various property restrictions: all values from, some values from, restrictions on values. The set of built-in OWL datatypes was extended from strings and integers in OWL 1 to XML schema datatypes and various datatype restrictions. As the lack of keys in OWL 1 was recognized as an important limitation in expressive power keys were introduced into OWL 2. They may be defined on a list of object or data properties. Also, OWL 2 adds syntactic sugar to make some common patterns easier to write. Since these constructs are simply shorthands, they do not change the expressiveness, semantics, or complexity of the language.

There were three dialects defined in OWL 1: OWL DL, OWL Full and the syntactic subset OWL Lite. These dialects exposed their insufficiency for implementing tools working with OWL ontologies [10]. To resolve the issues of OWL tools, three OWL profiles were proposed having properties useful for different kinds of computation. OWL 2 EL captures the expressive power used by large-scale ontologies, having many classes and properties. OWL 2 QL captures the expressive power of simple ontologies like thesauri and ER/UML languages; it is well suited for working with very large number of individuals, and where it is needed to access data directly via relational queries. OWL 2 RL is designed to implementation using rule-based technologies. All of these profiles have certain restrictions. Although we are oriented towards application of ontologies in information systems and storing in databases, we do not intend to link with OWL 2 QL profile as it has serious limitations and is suitable to applications requiring only very simple ontologies.

We already have discussed existing approaches for representing ontologies in databases [1–3, 8, 19, 20, 22] in [30] and concluded in the rationale of creating bidirectional, lossless, model-based transformations between ontologies and database schemas. Metamodels of ontology language and database schema serve for this purpose. We are following the OWL 2 metamodel [23] for representing ontology. For a relational database, we use a part of Common Warehouse Metamodel (CWM) [4] that currently is under extension to more powerful CWM 2.x named as ”Information Management Metamodel” (IMM) [15].

Also, we studied the approaches for inverse mapping – i.e. from relational databases to ontology [5, 6, 9, 13, 26] (the survey is given in [28]). We analyze aspects of RDB2OWL transformations for lossless OWL2RDB transformation as most of relational concepts may be mapped to ontology structures, but not every ontology concept may be directly mapped to a relational database.

Among all these methods we can distinguish two ultimate information-lossless cases: storing ontology and its instances in the same manner (one fact table) or storing ontology and its instances in different schemas in order to improve access to instances while retaining the capacity of reasoning over the ontology. The first transformation method does not lose information, but it uses advantages of relational databases just for saving many records and does not preserve the real relational structure. The schema is in a low normal form and the performance of using transformed information normally should be slow e.g. [20]. The similar method is highly powered in Oracle Semantic Storage as it is supported with the native functionality of the Oracle database [31], where functionality of triggers helps to reasoning in ontology (alike in business rule manipulation techniques e.g. [21]). The second approach is much more promising for ordinary database management systems (e.g. [1, 2]). However, existing methods of that kind do not cover the sufficient subset of ontology concepts.

Our OWL2RDB transformation combines direct mapping of ontology classes, properties and instances with representing axioms and restrictions in metadata tables. Herewith we consider the reverse transformation of ontology from a database for efficient reasoning that may be achieved by joint usage of ontology query language SPARQL [25] and relational database query language SQL [7]. Reasoners that use ontologies represented in
XML files usually extract ontology schema along with its instances into a main memory and perform all inference there [27]. Performing full reasoning in memory ensures the completeness of query results but it is unsuitable for large ontologies having many instances. In our case, only ontology structures are extracted into a memory and processed by the inference engine. Results of inference are used for accessing individuals by SQL queries obtained by converting fragments of SPARQL to SQL.

This process is optimized in PelletDB reasoner for Oracle DB 11g due to the powerfulness of Oracle [16]. If ontology is of the acceptable size, PelletDB loads both the schema and the instance data from Oracle DB, then computes and saves all inferences back to the Oracle Database, which can be queried without additional reasoning. When instance data are too large to fit into memory, PelletDb extracts only the schema, computes additional schema inferences, and saves these inferences in Oracle Database. Then it is possible to perform instance reasoning using the schema inferences. The combination of reasoning in-memory together with instance reasoning in database provides a viable means to achieving more complete inference and query results than either solution can offer alone. While there is no question about competing with Oracle, our approach scales for every size of applications and may be implemented in non-commercial database management systems.

3 OWL 2 Concepts and their Mapping to RDB Concepts

OWL 1 was mainly focused on constructs for expressing information about classes and individuals, and exhibited some weakness regarding expressiveness for properties. OWL 2 offers new constructs for describing properties, a richer set of datatypes and makes some common patterns easier to write. In this paper we analyse these OWL 2 concepts that in our opinion are the most useful for real world applications and can be transformed to relational database schemas. As basic mappings are similar to OWL 1 mappings presented in [29, 30], in this paper we are focusing on mappings of new constructs. As previously, we are combining mappings of OWL 2 concepts with RDB concepts and storing the problematic (in mapping sense) knowledge in metadata tables. For explaining the proposed mapping, we will use the extended excerpt of Wine Ontology as our example (Figure 1) where it is represented using UML OWL 2 profile [24] implemented in Protégé OWL2UML plug-in.

3.1 OWL Classes and Class Axioms

In OWL 2, classes and property expressions are used to construct class expressions that represent sets of individuals by formally specifying conditions on the individuals' properties; individuals satisfying these conditions are said to be instances of the respective class expressions. OWL 2 provides axioms that allow relationships to be established between class expressions (Figure 2).

When we are converting the OWL ontology description to relational database schema, we map each ontology class to a database table. As the name of an ontology class is unique in the whole ontology and instances have unique names, we create a primary key for each table by adding some suffix to the corresponding class name, e.g. “Id”, and the additional column by adding “Name” suffix to the class name for saving names of instances of the class. The fundamental taxonomic construct – the SubClassOf axiom, which allows to state that each instance of one class expression is also an instance of another class expression, is mapped to 1:0..1 relation.
in RDB. The subclass doesn’t need a column for saving names of its instances, because the instance of the subclass is also the instance of the super class and its name is already stated. These mappings for PortableLiquid, Wine, WineGrape, WineMaker and WineTaster classes of Wine Ontology example are presented in the upper part of Figure 3. We use our own UML profile for representing database schema where <<PK>>, <<FK>>, <<UK>> stereotypes mark primary keys, foreign keys and unique constraints; tags “id” mark names of foreign keys, and tags “uk” mark names of unique constraints.

![Figure 2. The OWL class descriptions diagram (23)](image)

The EquivalentClasses axiom defines that several class expressions are equivalent to each other, i.e. they have the same instances. The DisjointClasses axiom states that several class expressions are pair wise disjoint. The DisjointUnion class expression allows to define a class as a disjoint union of several class expressions and thus to express covering constraints. In our example, disjoint classes are the WineMaker and the CertificationCompany; they comprise one disjoint class group and we could be able to define a superclass of these classes e.g. “Company” as the disjoint union class if there were more disjoint groups. For preserving such information, we suggest saving all classes of the ontology in OWLClasses table with two main columns className, which is an auto increment identification number, and className, which saves the unique name of the class. This name is also the name of the corresponding table. Information about groups of disjoint and equivalent classes is saved in metatables OWLDisjointClasses and OWLEquivalentClasses. The groups of equivalent or disjoint classes also are represented in OWLEquivalentGroup and OWLDisjointGroup tables. Metatables for OWL 2 disjoint, equivalent and disjoint union classes are presented in the lower part of Figure 3.

### 3.2 OWL 2 Properties and Property Axioms

OWL 2 has two main categories of properties – object and data properties, and also annotation properties that may be useful for ontology documentation. Object properties relate individuals to other individuals. Data properties relate individuals to literals. We map the object property to the foreign key. Depending on the local cardinality of some class property and the object property is functional or not, one-to-many or many-to-many relation between tables of classes are created. In a case of many-to-many relation, an intermediate table must be created.

OWL 2 provides axioms for establishing relationships between object property expressions. The ObjectPropertyDomain and ObjectPropertyRange axioms can be used to restrict the first and the second individual, connected by an object property expression. The FunctionalObjectProperty and InverseFunctionalObjectProperty axioms define that each individual can have at most one outgoing or incoming connection of the specified object property expression respectively. The InverseObjectProperties axiom can be used to state that two object property expressions are the inverse of each other. The ReflexiveObjectProperty, IrreflexiveObjectProperty, SymmetricObjectProperty, AsymmetricObjectProperty, and TransitiveObjectProperty axioms define that an object property expression is reflexive, irreflexive, symmetric, asymmetric, or transitive. These axioms are represented in metatable “OWLObjectProperties” (Figure 3).

In OWL 2 there are two forms of object subproperties axioms. The basic form is SubObjectPropertyOf(OPE, OPE'). This axiom states that the object property expression OPE' is a subproperty of the object property expression OPE — that is, if an individual x is connected by OPE to an individual y, then x is also connected by OPE to y. E.g. in our example the class PotableLiquid has the object property HasMaker, and the class Wine has the object property HasWineMaker which is the subproperty of the property HasMaker. Information that one property is a subproperty of another property we save in the metatable OWLObjectProperties.

Another form of OWL2 object subproperty is ObjectPropertyChain. The axiom SubObjectPropertyOf(ObjectPropertyChain(OPE, ... OPE') OPE) states that, if an individual x is connected by a sequence of object property expressions OPE, ..., OPE, with an individual y, then x is also connected with y by the object property expression OPE. E.g. we have the class Wine and the object property isTastedBy with the range class
WineTaster. The class WineTaster has the object property worksForCompany with the range class CertificationCompany. We can declare the axiom SubObjectPropertyOf(ObjectPropertyChain(a:isTastedBy a:worksForCompany) isVerifiedBy) that means if some wine is tasted by the taster who works for some certification company then this wine is verified by this company. ObjectPropertyChain axioms are represented in metatable OWLPropertyChain. (Figure 3). This table has links to the compound and component object properties and the sequence number of some component property in the property chain.

![Figure 3. Example of OWL wine ontology transformed into relational database](image)

At the last stage of transforming domain ontology into relational database, when whole schema is created, we must convert all assertions of classes and properties into records and fill the database. During this process object property chains can be used to gain some missing information about relations between objects. E.g. if we have both object property assertions isTastedBy and worksForCompany and the axiom SubObjectPropertyOf(ObjectPropertyChain(a:isTastedBy a:worksForCompany) isVerifiedBy) on some instance, we can create object property assertion and insert the appropriate value in the column isVerifiedBy of the table Wine automatically.

Ontology data properties relate individuals to literals. Functional data properties can be mapped to relational database columns of the tables corresponding to the domain classes of these properties. Because the OWL 2 was extended for representing ranges of data properties by the XML schema datatypes, we map XML schema datatypes to corresponding SQL datatypes. In a case of the data property is not functional or it has cardinality more than one, the data property is mapped to the additionally created table named by the data property name. This additional table has three columns – the auto increment identification number, the foreign key to the table of the corresponding domain class of this property and the value. The value column is SQL datatype corresponding to the XML schema datatype of the data property.
OWL 2 provides a new construct “HasKey” which allows keys to be defined for a given class. With this construct it is possible to give a list of object or data properties, which together identify resources of a given type. For example, if individuals of the class “Wine” are uniquely identified by data properties “wineName”, “wintageYear” and the object property “hasWineMaker”, then the OWL 2 axiom HasKey(Wine :wineName :wintageYear :hasWineMaker) states that each named instance of the class “Wine” is uniquely identified by this set of properties – that is, if two named instances of the class coincide on values for each of key properties, then these two individuals are the same.

For converting the OWL ontology description to the RDB schema, we map the “HasKey” axiom on some properties for the certain class to the uniqueness constraint of columns of the corresponding table. Depending on “HasKey” properties count (one or many), we create the unique key on the single column, or the multi column (combination of columns) unique index of the table.

3.3 OWL Restrictions

In OWL 2 class expressions can be formed by placing restrictions on object property expressions. The ObjectSomeValuesFrom(CE) class expression allows for existential quantification over an object property expression OPE, and it contains those individuals that are connected through an object property expression OPE to at least one instance of a class expression CE. The ObjectAllValuesFrom(CE) class expression allows for universal quantification over an object property expression OPE, and it contains those individuals that are connected through an object property expression OPE only to instances of a class expression CE. The ObjectHasValue(CE) class expression contains those individuals that are connected by an object property expression OPE to a particular individual a. Finally, the ObjectHasSelf(CE) class expression contains those individuals that are connected by an object property expression OPE to themselves.

When we are converting the OWL ontology description to the relational database schema we save this information in special metadata tables. ObjectAllValuesFrom, ObjectSomeValuesFrom and ObjectHasValue restrictions have their own metadata tables with column restrictedProperty which links to the table OWLObjectProperties. Metadata tables for ObjectAllValuesFrom and ObjectSomeValuesFrom restrictions also have column restrictionClass, which points to the table of the corresponding restriction source class (Figure 3). The ObjectHasValue restriction metadata table has the column “Value” for storing the value of the restricted resource of the corresponding property. Indication that object property has ObjectHasSelf restriction is saved in the column hasSelf of the OWLObjectProperties metatable.

Object property restrictions in OWL 2 can also be formed by placing restrictions on the cardinality of object property expressions ObjectMinCardinality, ObjectMaxCardinality, and ObjectExactCardinality that are saved in the metadata table OWLCardinality.

4 Querying OWL 2 Ontologies from Relational Databases

In this section we present the prototype for querying ontology, stored in a relational database according to the representation we have proposed. Usually, ontology reasoner (e.g. Pellet) reads ontology, including individuals, from a XML file (Figure 4). In our case, only ontology classes, their hierarchies, object and data properties, axioms and restrictions are extracted into a memory. Individuals are accessed by SQL queries obtained by converting fragments of SPARQL to SQL. The Ontology Database Integration component creates ontology model for the reasoner. This component analyses the database schema and metatables, builds the ontology model, rewrites SPARQL queries and executes SQL for obtaining results. The algorithm of transforming the database to ontology is based on the features of the previously described transformation from ontology into the database schema.

Figure 4. Common (a) versus enhanced (b) OWL reasoner

The following SPARQL query finds all potable liquids and their makers verified by the certain company:

```
select ?type ?drink ?maker where
 ?drink wine:isVerifiedBy wine:French_Certification_Company}
```

The presented query has four conditional clauses. According to the 1st clause, Pellet OWL Reasoner finds all subclasses of the PotableLiquid class according to predicate ?type rdfs:subClassOf wine:PotableLiquid,
where \texttt{FILTER} ensures finding of proper subclasses of the \texttt{PotableLiquid} (Pellet OWL Reasoner treats every class as a subclass of itself). The first clause of the query (\texttt{?type rdfs:subClassOf wine:PotableLiquid.FILTER(?type!= wine:PotableLiquid)}) returns the single record \texttt{Wine} and assigns it to the variable \texttt{?type}. SQL executes the remaining clauses. The second clause \texttt{?drink rdf:type ?type} rewritten to SQL finds all individuals of the class Wine. The 3rd clause \texttt{?drink hasMaker ?maker} finds all makers of all previously found individuals and assigns them to the variable \texttt{?maker}. The example of SQL query that finds the maker "Marieta" of the drink (\texttt{Wine}) "MariettaPetiteSyrah" (\texttt{WineId}=3):

\begin{verbatim}
SELECT MakerName, MakerId FROM PotableLiquid, Maker, Wine
WHERE PotableLiquid.hasMaker = Maker.MakerId and
    PotableLiquid.potableLiquidId = Wine.WineId and Wine.WineId = 3
\end{verbatim}

The last, 4th clause \texttt{?drink isVerifiedBy wine:French_Certification.Company} rewritten into SQL filters selected individuals by checking which of them is certified by the certain certification company. The sample data and results of the query are presented in Figure 5.

\begin{verbatim}
SELECT MakerName, MakerId FROM PotableLiquid, Maker, Wine
WHERE PotableLiquid.hasMaker = Maker.MakerId and
    PotableLiquid.potableLiquidId = Wine.WineId and Wine.WineId = 3
\end{verbatim}

Figure 5. Results from querying relational data representing Wine ontology

5 Conclusions and Future Work

In this paper we presented the mapping for transforming ontologies described in OWL 2 to relational database schemas. These mappings extend our previous transformation, oriented to OWL 1, with new concepts and offer more possibilities for representing the rich knowledge about a problem domain. Our OWL2RDB transformation combines direct mapping of ontology classes, properties and instances to database schema with representing axioms and restrictions in metadata tables. Our transformation is capable of the lossless representation of the chosen subset of ontology concepts in a database and the lossless retrieval of ontology schema from the database into ontology reasoning tools.

Our approach is well-suited for creating new databases from ontologies and creating ontologies for already existing databases. We argue that it is worth to story ontology structures in databases for ontology management purposes. We have tried a prototype of a tool for extracting ontologies from relational databases, satisfying our schema, and allowing the step-wise processing of SPARQL queries where SPARQL is used for querying ontology structures in a main memory and SQL is used for querying instances in the database.

Currently, we are working on the extension of our transformation tool and are willing to provide the transformations of the subset of OWL 2 concepts sufficient for representation of ontologies appropriate for applications of information systems. The OWL 2 QL profile that is oriented towards efficient implementation of tools working with ontologies stored in databases is unsuitable for real needs of information systems. OWL 2 QL profile has strong restrictions and is capable of working only with very simple ontologies when the actual applications require capturing business rules and transforming them to software code, integrating data from distributed resources, effectively communicating on the World Wide Web etc.

References


