Bidirectional Transformations in Database Evolution: A Case Study "At Scale"

Mathieu Beine
University of Namur
Namur, Belgium
math.beine@gmail.com

Nicolas Hames
University of Namur
Namur, Belgium
nicolas.hames@gmail.com

Jens H. Weber
University of Victoria
Victoria, Canada
jens@acm.org

Anthony Cleve
University of Namur
Namur, Belgium
anthony.cleve@unamur.be

ABSTRACT

Bidirectional transformations (BX) play an important role in database schema/application co-evolution. In earlier work, Terwilliger introduced the theoretical concept of a *Channel* as a BX-based mechanism to de-couple "virtual databases" used by the application code from the actual representation of the data maintained within the DBMS. In this paper, we report on considerations and experiences implementing such Channels in practice in the context of a complex real-world application, and with generative tool support. We focus on Channels implementing Pivot/Unpivot transformations. We present different alternatives for generating such Channels and discuss their performance characteristics at scale. We also present a transformational tool to generate these Channels.

Keywords

Bidirectional transformations, database evolution, schema-code co-evolution, performance

1. INTRODUCTION

Many of today's software applications are backed by data base management systems (DBMS), most of them using a relational data model. With increasing system complexity and changing requirements arises the need to adapt and evolve software applications to meet new objectives. In the context of database applications, adaptations may be performed at the database level (i.e., schema changes, data migration) or at the level of the software application (i.e., program code). Changes made at either level often necessitate changes at the other level in order for the overall system to keep functioning. The synchronization of adaptation at different levels is often referred to as the schema/program co-evolution challenge.

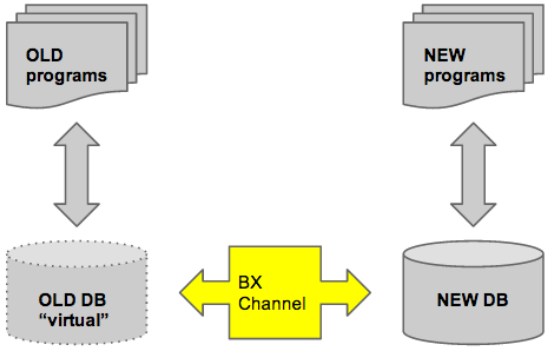

Figure 1: BX in DB/program co-evolution

Bidirectional transformations (BX) can be used as one way of addressing this co-evolution challenge. BX can be used to decouple the evolution of the database schema from the evolution of the program code, for example, by allowing changes to the database structure to be implemented while some programs can remain unchanged. In this case, any database access of the program code that uses the "old" schema is transformed to an equivalent database access using the new schema structure. Terwilliger [12] introduced the theoretical concept of a *Channel* to formalize this notion of transformations that translate application code queries to a "virtual database" structure to equivalent queries into the actual database implementation structure, cf. Figure 1.

From an engineering perspective, implementation of Channels in practice and at scale raises a range of design decisions and trade-offs. While previous authors have reported on experiences with implementing Channels (also referred to as wrappers) [3], such reports remain rare and often consider only small-scale applications and simple transformations only.

This paper presents empirical results from a large-scale industrial case study of engineering Channels to support the evolution of a complex medical information system. We focus on a couple of complex transformations, including *Pivot* and *Unpivot* and discuss their efficient implementation in practice. The Pivot and Unpivot operations can be described as rotation a table from a *1-column-per-attribute* to a

1-row-per-attribute representation and vice-versa. The reason why these operations are of particular interest from a software evolution point of view is that it is often beneficial to transform complex, sparsely populated table structures to a generic, more concise Entity-Attribute-Value (EAV) model. This reduces the complexity of the database schema as well as the programs accessing such data. The following case study will provide an example for such a transformation.

1.1 Case Study: OSCAR EMR Software

The application case study used in our work is a real-world medical information system called *OSCAR* used in primary health care in hundreds of clinics in Canada [8]. OSCAR has evolved over more than a decade and its current database includes more than 450 tables. The OSCAR database consists of well-populated "core" table structures that store information about patient demographics, allergies, medications, active problems etc., as well as more specialized, "satellite" table structures that store information for specific types of encounters and patient situations in primary care. These more specialized table structures are often associated with elaborate electronic *forms* that are filled out by the clinician on certain types of patient encounters. Due to the broad spectrum of different conditions that patients may have, these tables may have thousands of columns but any given data record (the actual data in each row) may only be populated sparsely (i.e., many `null` values).

The data in these large, sparsely populated tables are more adequately represented in a *1-row-per attribute* format, also called Entit-Attribute-Value (EAV) format, in order to save space and simplify program access. To see how the EAV representation might simiplify program access, consider a program that exports all encounter information, including all forms that may exist for a given patient. Using the EAV model, such a program will not need to query a large set of different tables and probe the existence of values in each column.

OSCAR's schema includes more than 60 form tables that can be transformed to a generic model in this way. This situation is by no means unique to our case study systems. Other systems and vendor products we have been working with show a similar structure and are expected to benefit from similar schema transformations.

1.2 Contributions and overview

This paper makes two main contributions. Firstly, we discuss implementation decisions and trade-offs related to the implementation of pivoting and unpivoting Channels at scale in the context of a real-world, industrial application. Secondly, we present a transformational tool for generating such Channel implementations in support of database schema evolution.

The rest of this paper is structured as follows. The following section provides an overview over research work related to our topic. Section 3 defines the transformations used in our work in more detail. Section 4 discusses implementation alternatives and trade-offs realizing the pivoting Channel. Section 5 presents the transformational tool we implemented for generating Channel implementations. Section 6 presents quantitative results from studying the performance of Channel implementations. Finally, Section 7 offers conclusions and directions for current and future work.

2. RELATED WORK

Most software must continue to adapt to fit changing requirements to remain useful. Database applications are no exception. In the context of database evolution, we are primarily interested in changes that involve the database schema definition. Evolution of program code that does not impact the database is out of scope for this paper and subject to a broad spectrum of research on different aspects of software evolution and reengineering. The reader may refer to [10] for a general overview.

Changes to the schema definition of a database application usually (but not always) require updates to application code (programs) that use the database as well as updates to the actual data instances. These two kinds of updates are commonly referred to as *application migration* and *data migration*, respectively [9]. A common strategy for adapting application programs to database changes is to use so-called *wrappers*, i.e., programs that "hide" the database changes from the application program by effectively translating queries (and updates) of the "old" database to equivalent accesses to the "new" (changed) database [13]. Application programs adapted in such a way can remain unchanged. Conversely, wrappers can also be used to accommodate evolution in application programs, while leaving the database implementation unchanged. In that case, the wrapper will transform queries (and updates) from newly developed or evolved programs (requiring a modified database structure) to accesses on the "old" database implementation.

In practice, large-scale database applications that have evolved over longer periods of time often have to support a combination of wrappers for forward as well as backward compatibility of different versions of programs with different versions of databases.

Of course, the question as to whether or not it is possible to create a wrapper that adapts a particular program (version) to a particular version of a schema depends on the nature of changes made in the schema (or the program). Schema changes are usually formalized in terms of transformation functions and categorized as information capacity preserving, -augmenting, -reducing. Thiran et al. [13] have proposed a semi-automatic approach for generating wrappers from composition of well-defined schema transformations. They report experiences with wrapping a small and a medium size system, but do not consider performance characteristics or more complex transformations, such as the ones discussed in this paper.

In later work, Terwilliger extends the concept of database wrappers to that of so-called *Channels* [12]. The latter not only transforms data queries and manipulations (queries / inserts / updates) from a "virtual" database to the real datatabase, but also transforms schema manipulations in a similar manner. In other words, from the point of view of an application program, a Channel should be indistinguishable from a "real" database. While Terwilliger discusses many of the same transformations as Thiran et al. (e.g., table partitioning and merging), he also discusses *Pivot* and *Unpivot* transformations. These two tranformations are of particular importance for database evolution and studies of their efficient implementation in auto-generated wrappers (or channels) with "at-scale" systems are scarce. We therefore concentrate on these transformations in this paper.

Research on BX has not been confined to the domain of databases, but other communities such as software engi-

neering, programming languages and graph transformations have studied BX based on different theoretical frameworks. Czarnecki et al. [4] provide an overview and comparison of BX theories accross disciplines. A theoretical framework of particular influence on the BX community has emerged from programming language domain, namely the lens framework [6]. In its most basic form, BX are considered as pairs of functions commonly referred to as $get : S \rightarrow V$ and $put : S \rightarrow V \rightarrow S$, where the first one produces a view V on a source data structure S and the second one updates the source data structure S according to any changes made to V. The special case of applying *put* to an empty source model in S is also referred to as *create*, i.e, $create(v) \equiv put(v, \emptyset)$. While our work on Channels is not formally based on the theory of lenses, we will informally adopt the *get*/*put*/*create* terminology framework [11] to describe the transformation that creates a virtual database (get) for the purpose of legacy program access and propagates any updates to that virtual DB to the real database (put) (cf. Figure 1).

Research on generating bidirectional channnels (or wrappers) is related to the well-known *view-upate problem* in databases [1]. Bohannon et al. [2] present *relational lenses* as an attempt to adapt the lens framework to address the relational view update problem from an algebraic perspective. They propose a new programming language to formalize view update policies in terms of lenses and define formal laws on well-behavedness of these lenses. While related, our work on database schema evolution has a different objective. Rather than programming BX, we are interested in automatically generating BX Channels as a side effect of applying schema redesign transformations during the process of evolving and refactoring database applications.

3. TRANSFORMATION DEFINITION

In this section, we provide definitions for the primitive and composite transformation used in this paper.

3.1 Pivot and Unpivot

The *Pivot* operator (T'=PIVOT(T,A,V)) transforms a table T in generic key-attribute-value form into a form with one column per attribute. Column A must participate in the primary key of T and provide the names for the new columns in T', populated with data from column V. The resulting table is named T'. The formal definition given for the Pivot operator using relational algebra is presented below and based on [12]. Readers who are unfamiliar with relational algebra are referred to [7] for a primer.

$$\not\boxtimes_{\mathbf{C};A;V} T \equiv$$

$$(\pi_{columns(T)-\{A,V\}} T) \bowtie (\rho_{V \rightarrow C1} \pi_{columns(T)-(A)} \sigma_{A=C1} T)$$

$$\bowtie ... \bowtie (\rho_{V \rightarrow Cn} \pi_{columns(T)-\{A\}} \sigma_{A=Cn} T)$$

$$for \;\; C1, ..., Cn = \mathbf{C} = \delta(\pi_A(T))$$

Figure 2 shows the intermediate steps of the Pivot operation for an example table T with three columns. In this case, *Period* is the pivoting attribute A whose values will give rise to columns in the resulting table and *Price* provides the values for these columns. Figure 2 shows that intermediate tables are created for each arising attribute. The key for the resulting table T' will be all remaining columns in T (all columns other than A and V).

The *Unpivot* operator (T'=UNPIVOT(T,A,V)) is the inverse of the Pivot operator and transforms a table T from a one-column-per-attribute form into key-attribute-value triples, effectively moving column names into data values in new column A (which is added to the key) with corresponding data values placed in column V. The resulting table is named T'. The formal definition (given in [12]) for this operator in relational algebra is presented below.

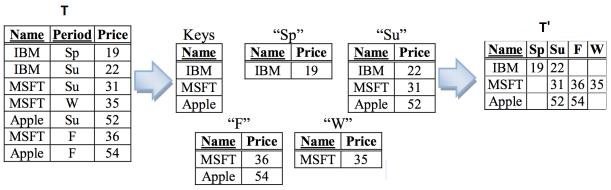

Figure 2: T'=PIVOT (T,Period,Price)

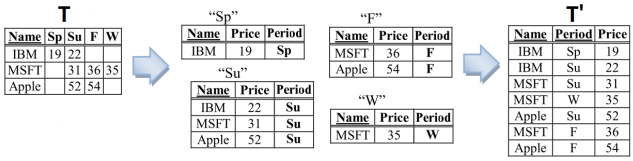

Figure 3: T'=UNPIVOT(T, Period, Price)

$$\not\boxtimes_{\mathbf{C};A;V} T \equiv$$

$$\bigcup_{c \in \mathbf{C}} (\rho_{C \rightarrow V} \pi_{columns(T)-(\mathbf{C}-\{C\})} \sigma_{C<>null}(T)$$

$$\times \rho_{1 \rightarrow A}(name(C)))$$

Figure 3 shows the intermediate steps of the Unpivot operation.

3.2 VPartition and VMerge

In practice, Pivot and Unpivot transformations are often used in composition with two other operators, commonly referred to as *VPartition* and *VMerge* in [12]. The $(T1, T2) = VPartition(T, f)$ operator splits a given table into two tables $T1, T2$, according to a total selection function f, which associates each non-key column with one of the two target tables ($T1$ or $T2$). Both resulting tables share the key columns of T. The $(T' = VMerge(T1, T2))$ operator is the inverse of the $VPartition$ operator and reconstructs a single table using two tables sharing a common primary key. The formal definition of VPartition and VMerge is straight forward based on projection and joins in the relational algegra, respectively, and omitted here.

3.3 Complex transformations: create/get/put

Complex transformations (and the Channels that implement them) can be composed by concatenations of primitive ones, such as the ones defined above. The composite transformation we will focus on in our case study combines VPartition and Unpivot to transform database structures in one-column-per-attribute format into equivalent structures into an Entity-Attribute-Value (one-row-per-attribute) format (akin to *create* and *put* in the lens framework). The inverse transformation composes Pivot and VMerge to re-

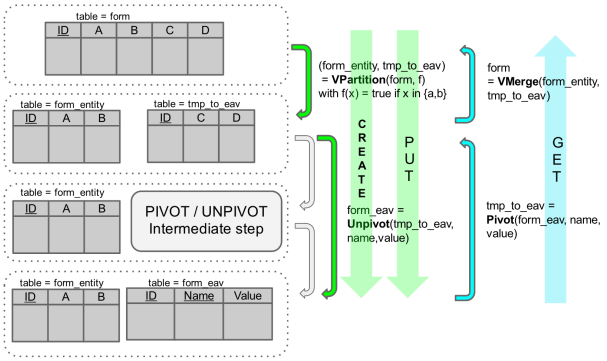

Figure 4: Composite BX - create/get/put

construct the original structure (akin to *get*). Figure 4 illustrates these transformations with a graphical example.

4. CHANNEL IMPLEMENTATION

Different strategies and techniques can be applied when implementing Channels for the above transformations. This section describes and compares several such alternatives and presents a novel technique referred to as the "coalescing approach". For this discussion, we assume that the database management system (DBMS) used does not have built-in operators for Pivot and Unpivot transformations. Indeed, most current DBMS still lack these operators.

In most of the examples below, we present some SQL pseudo-code in order to help the reader to fully understand the theory. All the examples are based on the tables from the schema available in the Figure 4.

4.1 Implementing "Create"

In our application domain of database evolution, *create* is mainly used for the data migration task, i.e., to transform data that conform to the "old" schema to equivalent data conforming to the newly evolved (transformed) schema. The amount of that data may be large in real-life applications.

This step can be implemented with a DB client program or directly within the DB server, in the form of a stored procedure. We implemented both alternatives. As expected, the first alternative is much less efficient. However, it has the benefit of being more platform independent. The algorithms are similar for both approaches and provided below.

4.1.1 The procedural approach

The procedural approach uses nested loops. The first loop inserts each entity in the entity table (the table containing the columns that will not be unpivoted, e.g., the entity keys and any columns that should remain in the original format). For each inserted entity, the second loop will be executed in order to insert the unpivoted attributes in the corresponding Entity Attribute Value (EAV) table. An example using the tables from Figure 4 is presented at Figure 5. Although it is a small example, it is easy for the reader to project this for such big tables as exist in real software systems.

4.1.2 The declarative approach

The declarative approach first inserts all the entities into the entity table and then it executes one "big" insert composed of unions of select to migrate all the unpivoted attributes into the corresponding table. The SQL pseudocode is given at Figure 6.

```
BEGIN
DECLARE id_var,A_var,B_var,C_var,D_var INTEGER;
DECLARE cur1 CURSOR FOR SELECT * FROM form;
OPEN cur1;
 read_loop: LOOP
 FETCH cur1 INTO id_var,A_var,B_var,C_var,D_var;
 IF (no more records){LEAVE read_loop;}
 INSERT INTO form_entity VALUES(id_var,A_var,B_var);
 ...
 IF(C_var is not null or C_var !=""){
 INSERT INTO form_eav VALUES(id_var,"C",C_var);}
 ... (for all the unpivoted attributes)
 END LOOP;
CLOSE cur1;
END
```

Figure 5: Procedural approach (Create)

```
BEGIN
 INSERT INTO form_entity SELECT id,A,B FROM form;
 INSERT INTO form_eav SELECT * from (
 SELECT id,name,value FROM (SELECT id,C AS value FROM
       form WHERE C IS NOT NULL),(SELECT "C" AS name
      FROM DUAL)
 UNION
 SELECT id,name,value FROM (SELECT id,D AS value FROM
       form WHERE D IS NOT NULL),(SELECT "D" AS name
      FROM DUAL))
END
```

Figure 6: Declarative approach (Create)

4.2 Implementing "Put"

The solution presented here is an implementation of the update channel transformation defined by Terwilliger [12]. The update channel transformation consists of three basic operations that are insert, update and delete. Database *triggers* are a natural solution for invoking these operations. Each time a "legacy" application inserts/updates/deletes the data in the virtualized (old) DB, a dedicated trigger executes the corresponding part of the *put* function. The listings at Figures 7, 8 and 9 sketch the SQL pseudocode for the insert, delete and update triggers in our example.

Of course, these triggered *put* functions may fail if some integrity constraints become violated due to concurrent updates, e.g., an insert fails if an item with the same key already exists in the target database. We do not further discuss concurrency issues in this paper, as it is assumed that legacy programs use transactions when accessing the virtualized database, and executing the Channel code is part of that

```
% SQL Code for Insert trigger
CRETE TRIGGER insert_form INSTEAD OF INSERT ON
     form_view
FOR EACH ROW BEGIN
INSERT INTO form_entity VALUES(NEW.id,NEW.A,NEW.B);
 ...
 IF(NEW.C IS NOT NULL){
 INSERT INTO form_eav VALUES(NEW.id,"C",NEW.C);}
 ... (for all the unpivoted attributes)
END
```

Figure 7: Insert trigger (Put)

```
%  SQL Code for Delete trigger
DROP TRIGGER IF EXISTS delete_form;
CREATE TRIGGER delete_form INSTEAD OF DELETE ON
    form_view
FOR EACH ROW
BEGIN
 DELETE FROM form_eav WHERE id=OLD.id;
 DELETE FROM form_entity WHERE id=OLD.id;
END
```

Figure 8: Delete trigger (Put)

```
% SQL Code for Update trigger
CREATE TRIGGER update_form INSTEAD OF UPDATE ON
    form_view
FOR EACH ROW BEGIN
UPDATE form_entity SET B=NEW.B,A=NEW.A where id=NEW.
    id;
...
IF(NEW.C not like OLD.C){
 IF(OLD.C==""){
  INSERT INTO form_eav VALUES(NEW.ID,"C",NEW.C);
 }ELSEIF(new.id is null){
  DELETE FROM form_eav WHERE name="C" AND ID=NEW.ID;
  ELSE
  UPDATE form_eav SET value=NEW.C WHERE name="C" AND
      ID=NEW.id;
 }
}
... (for all the unpivoted attributes)
END
```

Figure 9: Update trigger (Put)

transaction (and potentially aborts it in case of conflicts). Pessimistic strategies (locking) may be used to avoid such inconsistencies at the cost of limiting concurrency. However, locks applied on the virtualized DB should be propagated through the Channel to the actual DB to be effective. Terwilliger's current model of Channels does not cover the propagation of locks, nor does our implementation of Channel transformations [12]. We will address this limitation in future work.

4.3 Implementing "Get"

The *get* function recreates the old "virtual" database for the legacy programs to use. To implement the get function, we first followed the formal definition presented in Section 3.1. However, we found scalability problems with this solution. We therefore present a second implementation right after to avoid these problems.

4.3.1 The join approach

In order to allow "legacy programs" to keep working on the virtualized "old" database, we defined a set of queries that can be used to define views on the database. We implemented DML-SQL triggers to handle the usual CRUD operation on the new schema through the "virtual schema". This implementation is based on the theorical solution described in [12].

- *The Join approach* The join approach creates for each column that was unpivoted from the original table, an intermediate table containing the key-attribute-value triple for all the non-null values in the original table. Those intermediate tables are then joined together in order to create our "virtual schema". A pseudocode example is given at Figure 10.

  This approach is theoretically perfect. However, at scale, we found that DBMS run into the problem of the maximum-joins-per-query limit. The DBMS used in our case study application (MySQL) allows 61 joins per query. Some DBMS have higher limits, such as Microsoft's SQLServer, which accepts up to 256 joins. However, some of the tables in our case study have thousands of columns, which would require thousands of joins, clearly exceeding such a limit.

```
SELECT a.id,a.A,a.B,a1.value as C,a2.value as D
FROM form_entity a
LEFT OUTER JOIN form_eav as a1
 ON a1.id=a.id
 AND a1.name="C"
LEFT OUTER JOIN form_eav as a2
 ON a2.id=a.id
 AND a2.name="D";
```

Figure 10: Join approach (Get)

- *The join approach revisited* A solution to face the maximum-joins-per-query limit is to split the set of columns to migrate into multiple subsets, execute the join approach for subsets having less than 61 columns (or whatever the join limit of the DBMS may be) and then joining all those subsets in the final table. This solution worked at scale but lacked in performance compared to the *coalescing* approach we will describe below.

4.3.2 The coalescing approach

We decided to design another solution to execute the Pivot operation. We refer to this solution as the "coalescing approach". The formal definition of the Unpivot operator for the coalescing approach is given below:

$$\vec{\not\square}_{\mathbf{C};A;V}(T) = \gamma_{(columns(T)-\{A,V\}),MAX(C_1),\ldots,MAX(C_n)}$$
$$((\bigcup_{c\in\mathbf{C}}(\rho_{value\to name(C)}(\pi_{value,id}(\sigma_{A=name(C)}(T)))))$$
$$\times(\rho_{1\to name(C'_1)}(null) \bowtie \ldots \bowtie \rho_{1\to name(C'_n)}(null)))$$
$$for\ \ C_1,\ldots,C_n = \mathbf{C}\ \ AND\ \ C'_1,\ldots,C'_n = \mathbf{C} - \{C\}$$

where $\mathbf{C}$ is the set of values on which to pivot (the set of attributes you want to pivot), A is the Pivot column (the column containing the values for the new column names) and V is the pivot-value column (the column containing the value for the attributes).

This operation can be decomposed in multiple intermediate steps that will be explained here.

For this operation, the query is executed with a group-by clause on the id on the EAV table.

First, the query selects each row in the EAV table (relation T) where the column A contains the name of a column that belongs to $\mathbf{C}$ (In the example given below, $C = \{(\text{columns(T)-\{A,V\}})\}$, i.e. all columns are pivoted.) and transform it from a 1-row-per-attribute to a 1-column-per-attribute representation. This will create as many rows as they are attributes for the given id in the EAV model.

```
SELECT id,A,B,C,D FROM
 (SELECT
  MAX(IF(a.name ='C',a.value,null)) AS 'C',
  MAX(IF(a.name ='D',a.value,null)) AS 'D',
  id FROM form_eav a GROUP BY id) AS grp, form_entity
       AS i WHERE i.id=grp.id;
```

Figure 11: Coalescing approach (Get)

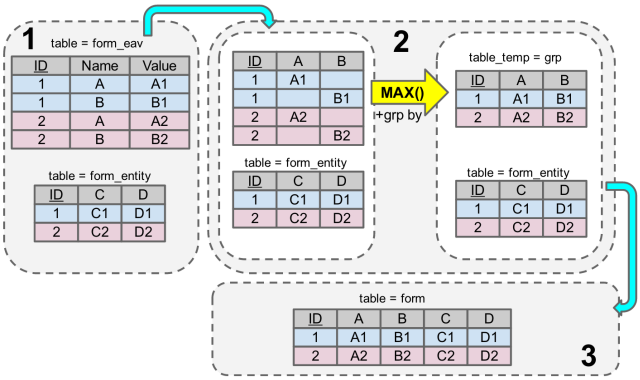

Figure 12: The coalescing approach

Then, for each row, the query will add columns that exists in the destination schema (all the values of **C**-A) and put a "null" value into those "joined" columns. This will produce a row having only one attribute with a non-null value per row and null-values for all the other columns.

Finally, the query executes an aggregate function (MAX) in order to "coaelesce" all the rows corresponding to same id into only one row.

This approach is significantly faster than the join approaches. We only have to execute one select for each entity and then execute only one join with the entity table, as shown at Figure 11. Figure 12 illustrates the coalescing algorithm described above. The example starts with the EAV model and reconstructs the original table, i.e., the virtual database. On the left (box 1), there is the EAV table and the table containing some columns kept in a 1-column-per-attribute representation. In the middle of the figure, box 2 presents the operation of pivoting the EAV table and joining the result with the entity table. The result of this box is the original table, or virtual database, presented in the box 3.

The aim of this MAX function is to coalesce all the rows that contains only one non-null value per row for each id into only one row per id, and so allow us to retrieve the original table. The example of Figure 13 (subschema of Figure 12) depicts the application of the max operator in this specific case. Here, the max function is used to retrieve the only "non-null" value for a specific column of a given id value.

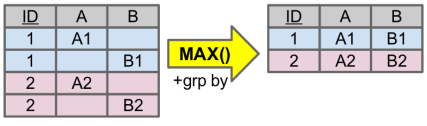

Figure 13: The MAX function

4.4 "Type-preservation" EAV model

One issue arising with transforming relational data into an EAV model (unpivoting) and back (pivoting) is the preservation of type information. Columns in the original table may use a large variety of different types. However, once cast into a joint EAV model, that type information may be lost, if it is not preserved. The implementation of type-preservation may complicate the resulting EAV model. We can consider different implementation alternatives summarized below. The three first alternatives have been described in [5].

1. The basic EAV schema. This schema store all the values in a single columns that usually uses a generic TEXT data-type. The original type information is lost.

2. The multi data-type EAV Schema. This schema use multiple tables, one for each data-type.

3. The hybrid EAV schema. This schema use multiple columns in the EAV table, one column for each data-type.

4. The Variant data-type. This schema use a variant data-type to store the different data-type. This solution has performance limitations and may not be offered in many DBMS systems. (It is not offered in MySQL, for example, the DBMS used by OSCAR.)

The choice we made for the Oscar case is to use a hybrid EAV schema with on table and multiple columns types. Since this may result in a potentially large number of columns, our transformation implementation generates an EAV schema to consider only those datatype that are really needed in the original tables.

This issue of type-preservation also implied that it is impossible to use the PIVOT and UNPIVOT functions that are sometimes defined in certain DBMS. As soon as we have to manage multiple column in the input for the pivot function or in the output for the unpivot function, we have to define our own implementation.

Another detractor of using PIVOT/UNPIVOT operators provided by some DBMS is that they are not well-defined and lack a unified semantics (see [14]). It is therefore not possible to predict what will be the output in specific cases as for example, a non-unique id for the pivot function.

5. TOOL SUPPORT

We developed a plug-in for DBMain(`www.db-main.eu`), an interactive database (re)engineering tool developed by the University of Namur and its spin-off company Rever. To date, DBMain offers rich support for database schema transformations, but does not generate Channels. Our plug-in extends DBMain with the capability of evolving database schemas based on the aforementioned transformations. DBMain generates the database definition of the newly evolved schema as well as all the code for the bidirectional Channel that allows legacy programs to run on the newly evolved database. We have experimented with different alternatives to implement the required transformations, particularly Pivot and Unpivot, as these operators are not provided as built-in primitives by most database management systems, or at least, not as we had to use it.

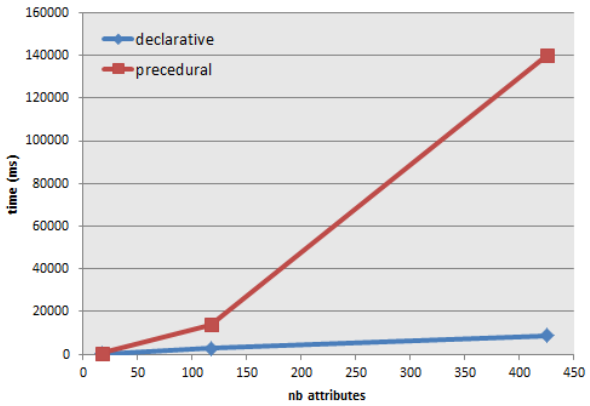

Figure 14: Create performance

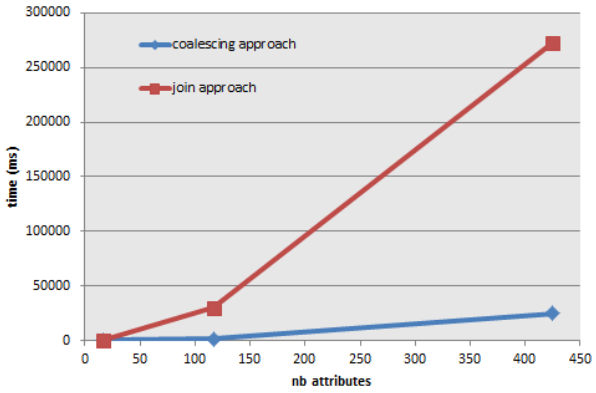

Figure 15: Pivot performance

The DBMain plugin can be used to pivot/unpivot tables from a DBMain schema (SQL to DBMain schema extraction also available in DBMain), one at a time or multiple at a time. The plugin then supports the migration of the existing data into the new EAV schema, generate the channel implementation (triggers), generate the view's, test the data migration, etc.

The tool support provided by the DBMain plugin is significant for two reasons, *correctness* and *scalability*. The first reason is related to the safety critical nature of healthcare information systems. There are strict requirements on the correctness of transformations and the ability for backwards compatibility of programs that use the "old" database structures. A tool that is capable of implementing schema transformations based on formally defined lossless transformations as well as generating code for Channels that can be used to automatically adapt "legacy programs" provides welcome assurance in this domain.

Secondly, the size of the Channel code generated by our plugin is considerable and writing this code by hand would be tedious and error prone. We found that in the best performing code (discussed below), each column in a transformed table gives rise to approximately 34 lines of code in the update ("put") function of the channel. A table of 1000 columns will therefore give rise to 34KLOC of channel code for the "put" direction alone.

6. IMPLEMENTATION COMPARISON

In order to evaluate the viability of our solution in a real world application we decided to perform some performance tests. We will provide here some performance measurements of the different implementations presented above.

First, we present a comparison of the performance for the data migration from the original model to the EAV format. This step corresponds to the implementation of the Channel *create* function. It is composed of a VPartition operation followed by the Unpivot operation. For these performance tests we decided to benchmark the data migration time on three different tables. We choose three tables from OSCAR containing 17, 117 and 425 columns. The following chart gives an overview of the time needed to migrate 1000 records from the original table to the EAV table.

Figure 14 shows the different performance characteristics of the declarative implementation and the procedural methods.

The performance gap between the two unpivoting methods can be understood by taking a look on the SQL query. The procedural statement executes a query for each value of each line that have to be unpivoted to insert all the field value one by one. The DBMS query optimizer is not able to optimize this iterative loop. In the other hand, the declarative approach executes only one insert query. This query is composed of one sub-query per attribute, but is not directly dependent of the number of rows contained in the original table, even if it will impact the data set size. The DBMS query optimizer can optimize and execute this single nested query more efficiently.

We also took measurements on the view reconstruction query ("get"). First, a Pivot operation is executed and then a VMerge is applied on the result of the Pivot operation with the table that contains the attributes kept in the "classical" relational form. We present in the Figure 15 the time to pivot/merge the table for the join approach and the coalescing approach. For these measurements we took the same tables as above and measured the time needed to perform a select query on the EAV model containing 1000 entities.

Comparing the performances of the two pivoting methods, we see that the first method uses a lot of joins (costly database operation), by creating one temporary sub-table per pivoted attribute. The second approach (coalescing approach) performs a unique select query that retrieves a huge result-set, then manipulates it to pivot the data. This method performs no join nor any costly operator. It only uses a single select with some conditions and is therefore faster.

7. CONCLUSION

Database evolution raises the challenge of co-evolving all program code that uses the database, unless we can put in place "adapters" that allow programs to remain unchanged and use the database in its "old format". Bidirectional transformations (BX) and Channels implementing BX can play an important role in keeping legacy applications running while evolving the database to a more suitable structure. In this paper, we have reported on experiences of generating Channels for an industrial case study "at scale". In particular, we focus on Channels involving transformations between traditional relational structures (one column per attribute) and generic project data structures (one row per attribute).

Implementation alternatives of these transformations have not been studied at scale to date. We present performance and salability aspects related to diffent implementation techniques and propose a novel approach for implementing the Pivot operator, referred to as the coalescing technique. We developed a plug-in for DBMain that extends the database reengineering tool with capabilities of generating Channel implementation code. Our future work is on researching ways in which Channel transformations can be implemented by means of object-relational mapping descriptions. Current object-relational middleware does not have support for complex transformations, such as Pivot and Unpivot, and would have to be extended to implement such Channels.

8. REFERENCES

[1] F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Trans. Database Syst.*, 6(4):557–575, Dec. 1981.

[2] A. Bohannon, B. C. Pierce, and J. A. Vaughan. Relational lenses: A language for updatable views. In *Proceedings of the Twenty-fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '06, pages 338–347, New York, NY, USA, 2006. ACM.

[3] A. Cleve, J. Henrard, D. Roland, and J.-L. Hainaut. Wrapper-based system evolution - application to CODASYL to relational migration. In K. Kontogiannis, C. Tjortjis, and A. Winter, editors, *Proceedings of the 12th European Conference in Software Maintenance and Reengineering (CSMR 2008)*, pages 13–22. IEEE Computer Society, 2008.

[4] K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. F. Terwilliger. Bidirectional transformations: A cross-discipline perspective. In *Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations*, ICMT '09, pages 260–283, Berlin, Heidelberg, 2009. Springer-Verlag.

[5] S. El-Sappagh, S. El-Masri, A. M. Riad, and M. Elmogy. Electronic health record, data model optimized for knowledge discovery. *International Journal of Computer Science issues*, 9, 2012.

[6] J. N. Foster, A. Pilkiewicz, and B. C. Pierce. Quotient lenses. *ACM Sigplan Notices*, 43(9):383–396, 2008.

[7] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2 edition, 2008.

[8] M. Gobert, J. Maes, A. Cleve, and J. Weber. Understanding schema evolution as a basis for database reengineering. In *Proceedings of the 29th IEEE International Conference on Software Maintenance (ICSM 2013)*. IEEE Computer Society, 2013.

[9] J.-L. Hainaut, A. Cleve, J. Henrard, and J.-M. Hick. Migration of legacy information systems. In *Software Evolution*, pages 105–138. Springer Berlin Heidelberg, Jan. 2008.

[10] T. Mens and S. Demeyer, editors. *Software Evolution*. 2008.

[11] J. Terwilliger, A. Cleve, and C. Curino. How clean is your sandbox? : Towards a unified theoretical framework for incremental bidirectional transformations. In *Proceedings of the 5th International Conference on Model Transformation (ICMT 2012)*, volume 7307 of *Lecture Notes in Computer Science*, pages 1–23. Springer, 2012.

[12] J. F. Terwilliger. Bidirectional by necessity: Data persistence and adaptability for evolving application development. In *GTTSE*, pages 219–270, 2011.

[13] P. Thiran, J.-L. Hainaut, G.-J. Houben, and D. Benslimane. Wrapper-based evolution of legacy information systems. *ACM Trans. Softw. Eng. Methodol.*, 15(4):329–359, Oct. 2006.

[14] C. M. Wyss and E. L. Robertson. A formal characterization of pivot/unpivot. In *Proceedings of the 14th ACM International Conference on Information and Knowledge Management*, CIKM '05, pages 602–608, New York, NY, USA, 2005. ACM.