Adapting Queries to Database Schema Changes in Hybrid Polystores

Fink, Jerome; Gobert, Maxime; Cleve, Anthony

Published in:
Proceedings of the 20th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2020)

DOI:
10.1109/scam51674.2020.00019

Publication date:
2020

Document Version
Peer reviewed version

Link to publication

Citation for pulished version (HARVARD):
Fink, J, Gobert, M & Cleve, A 2020, Adapting Queries to Database Schema Changes in Hybrid Polystores. in *Proceedings of the 20th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2020).*, 9252014, Proceedings - 20th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2020, IEEE Computer Society Press, pp. 127-131. https://doi.org/10.1109/scam51674.2020.00019

Adapting Queries to Database Schema Changes in Hybrid Polystores

Jérôme Fink
PReCISE, Namur Digital Institute
University of Namur, Belgium
jerome.fink@unamur.be

Maxime Gobert
PReCISE, Namur Digital Institute
University of Namur, Belgium
maxime.gobert@unamur.be

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

***Abstract*—Database schema change has long been recognized as a complex, time-consuming and risky process. It requires not only the modification of database structures and contents, but also the joint evolution of related application programs. This co-evolution process mainly consists in converting database queries expressed on the source database schema, into equivalent queries expressed on the target database schema. Several approaches, techniques and tools have been proposed to address this problem, by considering software systems relying on a single database. In this paper, we propose an automated approach to query adaptation for schema changes in *hybrid polystores*, i.e., data-intensive systems relying on several, possibly heterogeneous, databases. The proposed approach takes advantage of a conceptual modeling language for representing the polystore schema, and considers a generic query language for expressing queries on top of this schema. Given a source polystore schema, a set of input queries and a list of schema change operators, our approach (1) identifies those input queries that cannot be transformed into equivalent queries expressed on the target schema, (2) automatically transforms those input queries that can be adapted to the target schema, and (3) generates warnings for those output queries requiring further manual inspection.**

***Index Terms*—database schema evolution, query adaptation, hybrid polystores**

I. Introduction

In the last years, an increasing number of organizations have been considering NoSQL database engines as a migration target for existing information systems or as a platform for planned future systems. This trend is motivated by the sake of high performance and availability. However, while small-scale data loss or temporary inconsistency can be tolerable for some subsets of the data managed by an organization, for certain types of business-critical data such limitations are unacceptable. The consensus that seems to be emerging from the relational/NoSQL debate indicates that the two types of systems address substantially different classes of problems and that they should be selected accordingly. As a result, organizations increasingly need to use both types of databases in parallel, with an unavoidable data overlap between them.

As a consequence, today's data-intensive systems often exhibit software evolution requirements that cross-cut the problem domains of relational and NoSQL systems. Unfortunately, there is a lack of methodological guidance, dedicated techniques, and tool support for evolving such heterogeneous datastores, also called *hybrid polystores*, that would bridge different types of data stores in the context of schema evolution.

This paper contributes to filling this gap, by presenting a tool-supported approach to the adaptation of database queries under hybrid polystore schema evolution. This approach takes advantage of a conceptual modeling language for representing the polystore schema and considers a generic query language for expressing queries on top of this schema. Given a source polystore schema, a set of input queries and a list of schema change operators, our approach (1) identifies those input queries that cannot be transformed into equivalent queries expressed on the target schema, (2) automatically transforms those input queries that can be adapted to the target schema, and (3) generates warnings for those output queries requiring further manual inspection.

The remainder of the paper is structured as follows. In Section II, we position the novelty of our approach with respect to related literature. Section III summarizes the technical background of our approach, which builds on existing polystore modeling and query languages. We present our query transformation approach and its implementation in Section IV and an example usage scenario in Section V. Section VI concludes the paper and anticipates future work.

II. Related Work

The adaptation of client application programs under database schema evolution is a complex process addressed by several existing approaches, techniques and tools.

The *PRISM* workbench [1] provides an integrated support to *relational* schema evolution. It includes (1) a language for the specification of Schema Modification Operators (*SMOs*) for relational schemas, (2) impact analysis tools that evaluate the effects of such operators, (3) automatic data migration support, and (4) translation of old queries to work on the new schema. Query adaptation derives from the *SMOs* through SQL view generation and query rewriting techniques.

The *2LT* project [2] aims to formalize and to provide generic support for *two-level transformations*, which involve a transformation on the level of types with transformations on the level of values and operations. The solutions offered by the *2LT* project combine existing techniques of data refinement, typed strategic rewriting, point-free program transformation

This work was supported by the European Union H2020 research and innovation programme under the TYPHON project (#780251), and by the F.R.S-FNRS and the FWO under the EOS SECO-ASSIST project (#30446992).

and advanced functional programming. This generic approach revealed to be applicable to the coupled transformation of database schemas, data instances, queries and constraints.

Bidirectional transformations [3] can also be used to decouple the evolution of the database schema from the evolution of the queries, by allowing changes to the schema to be implemented while some queries can remain unchanged. In particular, the concept of *Channel* [4] was introduced to formalize transformations that translate application code queries to a “virtual” database schema to equivalent queries into the actual schema.

Stonebraker *et al.* [5] discuss two possible ways of co-evolving database schemas and related programs. A first way is the *data-first* way. It consists in first evolving the database schema, keeping it in the third normal form (3NF) and then adapting the application program regarding those changes. In real-world companies, this is almost never applied [6], due to potential higher cost and difficulties of program maintenance. Therefore the *application-first* strategy is favored. It consists of mitigating or even avoiding application code changes. Those two approaches do not constitute fully-satisfying solutions. The first one leads to program decay as applications may not correctly be adapted to the schema changes. The second leads to a database decay as data may be duplicated and thus the schema may become less and less conform to the 3NF.

To avoid such problems, Stonebraker *et al.* recommend to add an intermediate layer accessing the database(s) and to make application programs manipulate data through this layer. In the case of schema evolutions, the database access API would not change from the programs’ point of view, but only the implementation of the API functions would change, under the responsibility of the database administrators. An implementation of this approach has been proposed in [7], where the authors propose an automated derivation of a relational database from a conceptual schema and the automated generation of a data manipulation API providing programs with a *conceptual* view of the relational database.

All approaches discussed above consider schema-program co-evolution for software systems relying on a *single* database. In this paper we address the same problem, but for the more complex case of *hybrid polystores*. In this context, the presence of heterogeneous data models and query languages increases the difficulty of the task. Our approach is, however, inspired by previous approaches and combines their best ingredients, namely (1) the use of an intermediate layer between the programs and the polystore databases, (2) the use of a generic modeling language for representing the polystore schema at a *conceptual* level, (3) the use of an intermediate query language for expressing polystore database queries, and (4) the specification of schema modification operators, each associated with source-to-source query transformation rules. In addition, our approach also supports the generation of status messages for each database query returned as output, indicating whether the query has been modified, has remained unchanged, has become invalid, or requires further manual inspection.

III. Background

Our query adaptation approach relies on previous contributions introduced by Kolovos *et al.* [8], in particular the TyphonML polystore modeling language and the TyphonQL polystore query language. Those languages allow one, respectively, to specify a unified conceptual schema for the various databases belonging to a hybrid polystore, and to rely on an intermediate query language enabling to query the whole polystore data with a single unified language. This section briefly presents those languages and highlights their benefits in the context of schema evolution in general, and query adaptation in particular.

A. TyphonML and TyphonQL

TyphonML [8] allows one to describe the structure and the placement of the data in a hybrid polystore. Listing 1 shows an example of TyphonML polystore schema.

```
// Conceptual section
entity Description{
    id : int
    description : string[500]
    product :-> Product[1]
}
entity Product{
    id : string[256]
    name : string[50]
    price: float
}
entity Order{
    id: int
    total_price : float
    products :-> Product[0..*]
    owner :-> User[1]
}
entity User{
    id : int
    name: string[50]
    cardNumber : string[16]
    orders :-> Order."Order.owner"[0..*]
}

// Physical mapping section
relationaldb RelationalDatabase{
    tables{
        table{
            ProductDB: Product
            index productIndex{
                attributes('Product.name')
            }
            idSpec('Product.name')
        }
    }
}
documentdb DocumentDatabase{
    collections{
        ReviewsDB: Review
    }
}

//Schema Modification Operators
ChangeOperators[
        merge entities Product Description as Product,
        split entity User to CreditCard attributes:[cardNumber]
    ]
```

Listing 1. TyphonML example

It is divided into three parts :

- **The conceptual part** describes the semantics of the polystore data, in terms of conceptual entities, their attributes and the relations between them;

- **The physical part** : describes the mapping between the conceptual entities and the physical components (tables, documents, graphs) of the polystore databases;
- **The change operators** : A list of schema modification operators (SMOs) to apply to the polystore schema. This is the entry point of the evolution process and they are provided by the user.

Once deployed, the polystore can be queried using the TyphonQL language [8]. The language proposes a unified concrete syntax for CRUD operations performed on the polystore. The TyphonQL query engine compiles those queries into native queries manipulating the actual databases of the polystore. This means that migrating a polystore entity from one database platform to another within the polystore, does not impact the related TyphonQL queries, which may remain unchanged. The TyphonQL engine maps dynamically the query towards the right DBMS based on the physical part of the TyphonML schema. Some examples of TyphonQL queries are given in Listing 2

```
// select the review entity of a specific product
from Product p select p.price where p.name == 'laptop'

// insert a product
insert Product {id: '298', name:
    'kettle', price: 5.23}

// update
update Product p where p.id == 563 set{name: 'laptop'}
```

Listing 2. Query examples

IV. Query adaptation

A. Evolution approach

Our approach considers a hybrid polystore that has been modeled using TyphonML and that is queried by means of TyphonQL. Without the TyphonML and TyphonQL abstractions, this process would require query transformation rules for each specific native query language of the polystore. This would result in a complex and hard to maintain implementation. The adoption of the TyphonQL unified query language allows us to mainly focus on the semantic changes applied to the polystore schema. Hence, our problem becomes: How to adapt TyphonQL polystore queries to an evolving TyphonML polystore schema?

The evolution process of an hybrid polystore consists into producing a target version of the polystore starting from a source model and applying a set of Schema Modification Operators. Figure. 1 depicts the evolution process involved in the query adaptation tool. It takes as input a source TyphonML schema (as described in Listing 1), a set of TyphonQL queries (as in Listing 2) running on the source schema and a set of Schema Modification Operators (SMOs section in Listing 1) to apply on the source schema. The output of the query adaptation process is the transformed set of queries running on the target TyphonML schema with their categories and annotations.

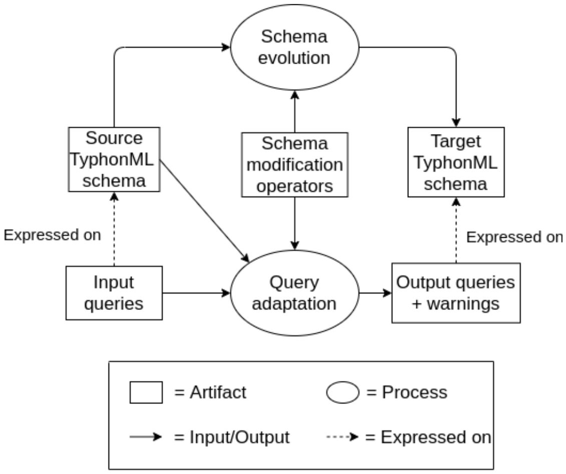

Fig. 1. Overview of the hybrid polystore evolution process

B. Schema Modification Operators and query adaptation rules

The *schema modification operators* (or SMOs) are evolution operations that manipulate objects of the TyphonML model, those objects include the **Entity**, the **Relation** or the **Attribute**. They are provided by the user and are the main entry point for the evolution process of the polystore. Our query adaptation approach is able to handle all changes that could happen on one of those three types of objects.

The evolution operators can be classified in three categories, depending on their semantic impact, i.e., the extent to which they preserve, augment or decrease the informational content of the polystore. A semantics-preserving schema modification ($\in S^{=}$), also called *schema refactoring*, does not impact the informational content of the polystore, but only the way the data is structured. This is the case, for instance, when renaming an attribute or when migrating an entity. *Semantics-augmenting* schema modifications (S^{+}) add informational contents to the polystore, for instance by adding an entity or an attribute. Conversely, *semantics-decreasing* schema modifications (S^{-}) remove some informational contents, e.g., when removing an entity or when restricting the cardinalities of a relationship.

$S^{=}$ operations can generally be propagated automatically to related queries and in some cases the queries may even be left unchanged. In the case of S^{-} or S^{+} operations, automated adaptation of queries is not always possible or needed.

To express those different situations, our query adaptation process distinguishes four possible categories of output queries:

- **UNCHANGED**: the input query has not been changed since it remains valid with respect to the target schema;
- **MODIFIED**: the input query has been transformed into an equivalent output query, expressed on top of the target schema;
- **WARNING** : the output query (be it unchanged or transformed) is valid with respect to the target schema, but it may return a different result set;
- **BROKEN**: the input query has become invalid, but it

Object	**Operation**	**Semantic class.**	**Create**	**Read**	**Update**	**Delete**
Entity	Add	S^+	U	U	U	U
	Remove	S^-	B	B	B	B
	Rename	$S^=$	M	M	M	M
	Merge	$S^=$	B	W	M	W
	Split	$S^=$	B	W	B	B
	Migrate	$S^=$	U	U	U	U
Attributes	Add	S^+	B	W	U	W
	Remove	S^-	B	B	B	B
	Rename	$S^=$	M	M	M	M
	Change type	$S^{-/+}$	W	W	W	W
Relations	Add	S^+	U	U	U	U
	Remove	S^-	B	B	B	B
	Cardinality change	$S^{-/+}$	W	W	W	W
	Rename	$S^=$	M	M	M	M

TABLE I
SCHEMA MODIFICATION OPERATIONS SUPPORTED BY OUR APPROACH AND THE WORST RESULT CATEGORY FOR EACH INPUT QUERY TYPE**U**: UNCHANGED, **M**: MODIFIED, **W**: WARNING, **B**: BROKEN

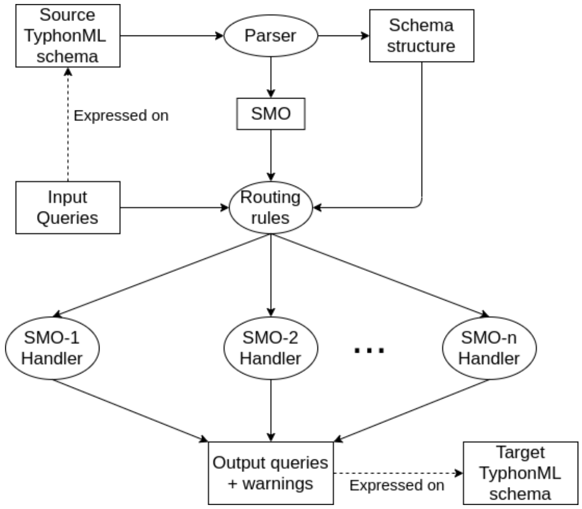

Fig. 2. Detailed view of the *Query Adpatation Process*

cannot be transformed into an equivalent query expressed on top of the target schema.

The queries labelled as BROKEN or WARNING are also annotated with a message explaining to the user which operator caused trouble. This helps the user to identify the issue and to manually adapt the query (or its context) to the new schema semantics, when needed/possible. Table I lists all the schema modification operators that we currently support and shows the worst result expected by this change on each type of queries (create, read, update and delete). For instance, when two entities are merged, the delete queries are annotated with a warning message as they would delete more information than before. When an attribute is removed, all queries that explicitly use the removed attribute are broken.

C. Implementation

Figure 2 goes into details of the query adaptation process. Firstly the tool parses the TyphonML schema provided as input to extract, on one hand, the current model structures and, on the other hand, the set of SMOs to apply. Secondly each operator is applied sequentially to each input query. The operator, the query and the schema are passed through routing rules that send them to the correct handler function according to the change operator processed. This routing is done by using extensively the pattern matching scheme of the Rascal Meta Programming Language [9]. Finally, the handler function produces the required transformations and query adaptation category. This complete structure makes it easy to support further additional SMO. The developer just has to add a new routing rule and a new handler function for the new change operator.

The resulting tool consists of an Eclipse plugin. Its code is open sourced[1] and an installation guide is also available along with demo scripts.[2]

V. EXAMPLE

In this section we illustrate the query adaptation process using a concrete example. Previously described Listing 1 represents a TyphonML schema containing evolution operators, in the Schema Modification Operator section, which describe the modification required by the user. The change operators are applied sequentially to the source schema. In this example, firstly the user wants to merge the *Description* entity into the *Product* entity and secondly wants to extract the attribute *cardNumber* of the entity *User* to a new entity called *CreditCard*.

Applying the merge and split operators consists of several atomic operations, respectively removing the *Description* entity, adding an attribute *description* to entity *Product* for the merge operation, the creation of a new relationship *to_CreditCard* in *User* and the creation of entity *CreditCard* for the split operation. The application of those operators results in a target TyphonML schema as shown in Listing 3.

```
entity Product{
    id : string[256]
    name : string[50]
    price: Real
    description : string[500]
}
entity Order{
    id: int
    total_price : float
    products :-> Product[1..*]
    owner :-> User[1]
}
entity User{
    id : int,
    name: string[50],
    orders :-> Order."Order.owner"[0..*]
    to_CreditCard :-> CreditCard[1]
}
entity CreditCard{
    cardNumber : string[16]
}
...
```

Listing 3. Target TyphonML Schema

[1] github.com/typhon-project/typhon-evolution

[2] Query adaptation examples

Let us consider the set of TyphonQL queries expressed in Listing 4. They consist of CRUD queries involving entities or attributes impacted by the provided SMOs.

```
/*1*/ from Product p, Description d select d.description
where d.product == p, p.id == "AZKIU",
/*2*/ from User u select u.cardNumber where u.name == "Doe",

/*3*/ insert User {id: 1, name: "Doe", cardNumber:"536864726"},

/*4*/ delete Product p where p.id = "EYIR",
/*5*/ delete User u where u.id == 5,

/*6*/ update Product p where p.id == "EYIR"
set{name: "Blender"},
/*7*/ update User u where u.id == 5
set {cardNumber:"5362637"}
```

Listing 4. Input queries expressed on the source TyphonML schema

The result of the query adaptation process is shown in Listing 5. For example Query 1 in Listing 4 selecting the *Description* attribute does not require the join condition anymore as this attribute in the target schema is now in the *Product* entity. Query 2 now needs a new join condition. Query 3 & 7 are now broken as *cardNumber* is not in *User* anymore. Query 5 is also marked as broken as *cardNumber* will not be deleted with the user anymore, in order to keep the same semantics two queries are required, one deleting the correct *CreditCard* entity and one deleting the *User*. This multi query adaptation constitutes a current limitation of this tool. Query 6 is not changed as it does not involve impacted attributes.

```
/*1*/ MODIFIED
#@ Product and Description merged @#
from Product p select p.description
where p.id == "AZKIU",

/*2*/ WARNING
#@ Entity User split into User, CreditCard @#
from User u, CreditCard c select c.cardNumber
where u.name == "Doe", u.to_CreditCard == c,

/*3*/ BROKEN
#@ Entity User split into User, CreditCard @#
insert User {id: 1, name: "Doe", cardNumber:"536864726"},

/*4*/ WARNING
#@ Product and Descriptions merged.
Delete will erase more information than before @#
delete Product p where p.id == "EYIR",

/*5*/ BROKEN
#@ Entity User split into User, CreditCard @#
delete User u where u.id == 5,

/*6*/ MODIFIED
#@ Product and Description merged @#
update Product p where p.id == "EYIR"
set{name: "Blender"}

/*7*/ BROKEN
#@ Entity User split into User, CreditCard @#
update User u where u.id == 5
set {cardNumber:"5362637"}
```

Listing 5. Output queries expressed on the target TyphonML schema

Using the adaptation classification (broken, warning, unchanged, modified) and its motivation messages above each query the user can make an informed decision of whether to use or not the output queries in his programs.

VI. Conclusion

In this paper, we propose a tool-supported approach, available through an Eclipse Plugin, which, given a conceptual polystore schema and a list of changes applied to this schema, is able to transform polystore queries into equivalent queries expressed on the evolved schema, when it is possible. If this is not possible, the user is provided with insight about what went wrong or what should be checked manually. The proposed approach is designed to work on a hybrid polystore, and relies on (1) a conceptual modeling language for representing the polystore schema, (2) a finite set of atomic schema modification operators to apply to this schema, (3) an intermediate polystore query language enabling the manipulation of the polystore data independently from the physical platforms (relational, NoSQL) where the data are actually stored. The current implementation has two main limitations. First, it is currently restricted to the transformation of each input query into another *single* query. The tool is not yet able to handle cases when an input query should be transformed into a script of multiple successive queries, e.g., when splitting an entity. Furthermore, the tool requires as input a set of polystore queries. It is not yet able to statically extract such queries from application programs or to infer and exploit related context information. This would require more sophisticated string analysis techniques (e.g., [10]), allowing the tool to deal with queries that are dynamically generated.

Our future work includes the further integration of our query transformation tool with a Java source code IDE, as well as the systematic evaluation of our approach based on industrial polystore evolution scenarios.

References

[1] C. A. Curino, H. J. Moon, A. Deutsch, and C. Zaniolo, “Update rewriting and integrity constraint maintenance in a schema evolution support system: Prism++,” *Proc. VLDB Endow.*, vol. 4, pp. 117–128, Nov. 2010.

[2] J. Visser, “Coupled transformation of schemas, documents, queries, and constraints,” *ENTCS*, vol. 200, no. 3, pp. 3–23, 2008.

[3] J. F. Terwilliger, A. Cleve, and C. Curino, “How clean is your sandbox? towards a unified theoretical framework for incremental bidirectional transformations,” in *ICMT 2012*, vol. 7307 of *LNCS*, pp. 1–23, Springer, 2012.

[4] J. F. Terwilliger, L. M. L. Delcambre, D. Maier, J. Steinhauer, and S. Britell, “Updatable and evolvable transforms for virtual databases,” *PVLDB*, vol. 3, no. 1, pp. 309–319, 2010.

[5] M. Stonebraker, D. Deng, and M. L. Brodie, “Application-database co-evolution: A new design and development paradigm,” *New England Database Day*, pp. 1–3, 2017.

[6] M. Stonebraker, D. Deng, and M. L. Brodie, “Database decay and how to avoid it,” in *2016 IEEE Int. Conf. on Big Data*, pp. 7–16, IEEE, 2016.

[7] A. Cleve, A.-F. Brogneaux, and J.-L. Hainaut, “A conceptual approach to database applications evolution,” in *Proc. of ER 2010*, vol. 6412 of *LNCS*, pp. 132–145, Springer, 2010.

[8] D. Kolovos, F. Medhat, R. Paige, D. Di Ruscio, T. Van Der Storm, S. Scholze, and A. Zolotas, “Domain-specific languages for the design, deployment and manipulation of heterogeneous databases,” in *11th IEEE/ACM Int. Workshop on Modelling in Software Engineering (MiSE)*, pp. 89–92, 2019.

[9] P. Klint, T. van der Storm, and J. J. Vinju, “Rascal: A domain specific language for source code analysis and manipulation,” *9th IEEE Int. Working Conf. on Source Code Analysis and Manipulation (SCAM)*, pp. 168–177, 2009.

[10] L. Meurice, C. Nagy, and A. Cleve, “Detecting and preventing program inconsistencies under database schema evolution,” in *IEEE Int. Conf. on Software Quality, Reliability and Security (QRS)*, IEEE, Aug. 2016.