

# The PRISM Workbench: Database Schema Evolution Without Tears

Carlo A. Curino <sup>#1</sup>, Hyun J. Moon <sup>\*2</sup>, MyungWon Ham <sup>\*3</sup>, Carlo Zaniolo <sup>\*4</sup>

<sup>#</sup>*Politecnico di Milano DEI, Milano, Italy*

<sup>1</sup>carlo.curino@polimi.it

<sup>\*</sup>*UCLA, Los Angeles, USA*

<sup>2</sup>hjmoon, <sup>3</sup>ham, <sup>4</sup>zaniolo@cs.ucla.edu

**Abstract**—Information Systems are subject to a perpetual evolution, which is particularly pressing in Web Information Systems, due to their distributed and often collaborative nature. Such continuous adaptation process, comes with a very high cost, because of the intrinsic complexity of the task and the serious ramifications of such changes upon database-centric Information System softwares. Therefore, there is a need to automate and simplify the schema evolution process and to ensure *predictability* and *logical independence* upon schema changes. Current relational technology makes it easy to change the database content or to revise the underlying storage and indexes but does little to support logical schema evolution which nowadays remains poorly supported by commercial tools. The *PRISM* system demonstrates a major new advance toward automating schema evolution (including query mapping and database conversion), by improving predictability, logical independence, and auditability of the process. In fact, *PRISM* exploits recent theoretical results on mapping composition, invertibility and query rewriting to provide DB Administrators with an intuitive, operational workbench usable in their everyday activities—thus enabling *graceful schema evolution*. In this demonstration, we will show (i) the functionality of *PRISM* and its supportive AJAX interface, (ii) its architecture built upon a simple SQL-inspired language of Schema Modification Operators, and (iii) we will allow conference participants to *directly* interact with the system to test its capabilities. Finally, some of the most interesting evolution steps of popular Web Information Systems, such as Wikipedia, will be reviewed in a brief “Saga of Famous Schema Evolutions”.

## I. INTRODUCTION

Information Systems are subject to a continuous pressure towards evolution, at the point that evolution has been recognized as a normal condition for an Information System. The era of “waterfall” design methodologies, which foresee an initial phase where all the system requirements are collected and frozen is undeniably over, especially in the context of Web Information System, which are typically characterized by a uninterrupted adaptation process due to the distributed and collaborative nature of their development and fruition.

The data management core represents one of the most critical portion of an Information System to be evolved [1]. In fact, while relational technologies achieved seamless evolution of the DB content, the problem of evolving a relational DB schema has been recognized as particularly pressing by Roddick [2] already in 1995, by denouncing the lack of appropriate tools and methods. Since then, there has been big

theoretical research effort to solve the underlying formal issues of schema mapping, mapping composition and invertibility, and query rewriting [3], [4], [5], [6]. This work produced elegant theoretical results, which, being based on formal logical mapping, lack practical usability required to provide the practitioner world with appropriate operational solutions.

*PRISM* has been designed and developed to bridge this gap, by harvesting recent theoretical advances on schema mapping [3], [4], [5] and query rewriting [6]. At the best of our knowledge, *PRISM* represents the most advanced system supporting relational schema evolution available to date. The system automates a very large portion of the query adaptation work, thus, providing an invaluable operational tool to support DB Administrators (DBAs) in their everyday activities. In this demonstration we will present the *PRISM* prototype and its intuitive AJAX-based Web Interface which guides the DBA through the phases of the evolution design process. This prototype represents also the starting point for a formal study of the tool usability we are planning to conduct.

*PRISM* addresses the two main challenges of Schema Evolution: *predictability* of the evolution process and *logical independence* of the evolution process. A DBA exploiting this workbench, represents the evolution step (s)he is willing to perform in terms of Schema Modification Operators (SMO), an operational language, which naturally captures the atomic operations to be used to evolve an existing schema.

Based on this representation the system solves the predictability challenge by means of a static analysis of the sequence of SMOs, from which it is possible to derive a crisp characterization of: i) information preservation, and ii) redundancy generation of the evolution being designed. The system bridges towards commercial DBMS<sup>1</sup> by automatically generating, given an SMO sequence, the corresponding SQL scripts that adapt the schema and migrate the data accordingly.

The second big challenge of Schema Evolution, namely evolution logical independence, is achieved in the system by deriving from the SMO sequence a logical mapping between schema versions required to automate the query rewriting. In particular *PRISM* translates SMOs into Disjunctive Em-

<sup>1</sup>The system have been tested on MySQL and DB2, the demo will focus on MySQL.

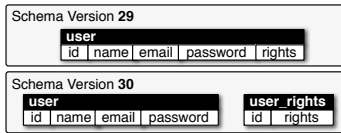


Fig. 1. A simple evolution step from the Wikipedia schema history: 29-30

bedded Dependencies (DEDs) and exploit the MARS [6] chase&backchase engine to rewrite user queries into equivalent ones under a different schema version. By automatically adapting legacy queries to the current schema *PRISM* effectively reduces the amount of manual work required during evolution. In our tests on the Wikipedia case study analyzed in [1], up to 70% of the queries fail, and thus requires manual rework, at each of the evolution step (over 170 for Wikipedia).

The query-rewriting technique we exploit significantly differs from data exchange approaches, e.g., Clío [7], for two main reasons: (i) mappings in *PRISM* are derived from SMOs, therefore, no automatic/manual matching of schemas is required, (ii) as opposed to rewrite queries, the data exchange approaches migrate the data under the schema queried by the users, this would be impractical in our scenario where the number of supported schema grows over time.

Among the most popular commercial tools trying to support schema change management it is worth to mention: DB2 Change Management Expert, Oracle Change Management Pack, MySQL Workbench for Schema Change, SwissQL DBChangeManager (MSSQL) and many others—see the *PRISM* homepage for details and links to these tools<sup>2</sup>. These tools only provide basic functionalities to support DBAs, lacking the advanced features offered by *PRISM*.

The rest of the paper is organized as follows: Section II briefly describes the Demonstration we are proposing, Section III provides a summary of the theoretical background, Section IV presents the system architecture and few implementation issues, and Section V draws our conclusions.

## II. THE DEMONSTRATION

To demonstrate the functionalities of the *PRISM* workbench, we exploit the Wikipedia DB schema as our running example. The actual demonstration is organized in three phases: (i) a brief introduction to the main system functionalities, in which the presenter shows a simple case of evolution highlighting the core components of the system, (ii) a “hands-on” phase in which the public is invited to *directly* interact with the system and test its capabilities, by designing evolution steps and exploring the resulting evolution, and (iii) what we like to call a “Saga of Famous Schema Evolutions”, a brief review of some of the most interesting and crucial steps in the schema evolution histories of popular Wikis and CMSs, among which the well known case of Wikipedia.

In the presentation of the system functionalities we will show how the system interface, shown in Figure 2, guides the user through the five steps of the design process:

(i) *Configuration*: in this phase, the user configures the DB

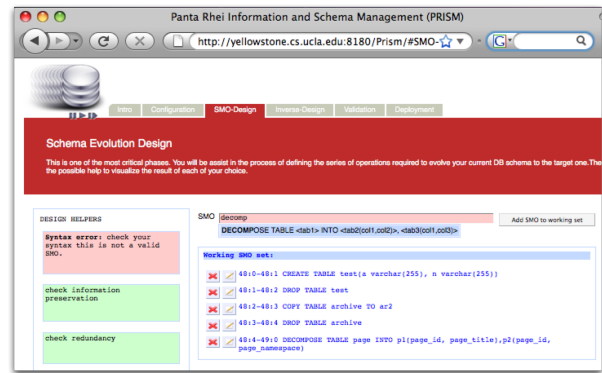


Fig. 2. *PRISM* AJAX Web Interface: screenshot of the SMO-Design Tab

to access providing appropriate credentials (demonstration on MySQL). The system loads the current schema and setup a repository for the schema history called Metadata DB .

(ii) *SMO Design*: in this phase, the user specifies a sequence of atomic changes in terms of SMOs. The system assists the user providing syntax, information preservation and redundancy checks. At every moment the user can inspect the (hypothetical) resulting schema, i.e., the system provide a support for *what-if* scenarios. Consider the example of Figure 1 presenting the Wikipedia schema evolution step 29 to 30, this evolution step is concisely captured by the SMO:

```
DECOMPOSE TABLE user INTO user_rights(id,
rights), user(id, name, password, email);
```

(iii) *Inverse Design*: in this phase, the system automatically generates an inverse sequence of changes—later needed to perform query rewriting, and thus legacy application support. In our running example the system automatically suggest the inverse SMO:

```
JOIN TABLE user, user_rights INTO user
WHERE user.id = user_rights.id;
```

The user can at any moment override the default system choices. The user intervention is required to disambiguate the few cases in which there exist more than one inverse for the forward sequence of changes. Given the inverse sequence the system automatically derives the DED mappings and compute percentage query support, by rewriting a set of queries logged during the system usage. The system provides feedback to the user showing the portion of the original schema and queries supported after each atomic schema change.

(iv) *Validation*: in this phase, the system provides a final level of check for the user by allowing the user to: (i) inspect the logical mapping between schema versions, and (ii) test the rewriting of queries. In our example the user might ask the system to rewrite the query: `SELECT * FROM user` which will be rewritten in: `SELECT * FROM user, user_rights WHERE user.id = user_rights.id`. At this point the user should have enough insight on the impact of the evolution (s)he designed to accept it or to roll back to a previous state and redesign part of it.

(v) *Deployment*: in this final step, the system generates

<sup>2</sup>See: <http://yellowstone.cs.ucla.edu/schema-evolution/index.php/Prism>

TABLE I  
SCHEMA MODIFICATION OPERATORS (SMOS)

SMO Syntax	Input rel.	Output rel.	Forward DEDs	Backward DEDs
CREATE TABLE R( $\bar{A}$ )	-	R( $\bar{A}$ )	-	-
DROP TABLE R	R( $\bar{A}$ )	-	-	$T(\bar{x}) \rightarrow R(\bar{x})$
RENAME TABLE R INTO T	R( $\bar{A}$ )	T( $\bar{A}$ )	$R(\bar{x}) \rightarrow T(\bar{x})$	$R_{V_i+1}(\bar{x}) \rightarrow R_{V_i}(\bar{x})$
COPY TABLE R INTO T	$R_{V_i}(\bar{A})$	$R_{V_i+1}(\bar{A}), T(\bar{A})$	$R_{V_i}(\bar{x}) \rightarrow R_{V_i+1}(\bar{x})$ $R_{V_i}(\bar{x}) \rightarrow T(\bar{x})$	$T(\bar{x}) \rightarrow R_{V_i}(\bar{x})$
MERGE TABLE R, S INTO T	R( $\bar{A}$ ), S( $\bar{A}$ )	T( $\bar{A}$ )	$R(\bar{x}) \rightarrow T(\bar{x}); S(\bar{x}) \rightarrow T(\bar{x})$	$T(\bar{x}) \rightarrow R(\bar{x}) \vee S(\bar{x})$
PARTITION TABLE R INTO S WITH <i>cond</i> , T	R( $\bar{A}$ )	S( $\bar{A}$ ), T( $\bar{A}$ )	$R(\bar{x}, cond) \rightarrow S(\bar{x})$ $R(\bar{x}, \neg cond) \rightarrow T(\bar{x})$	$S(\bar{x}) \rightarrow R(\bar{x}), cond$ $T(\bar{x}) \rightarrow R(\bar{x}), \neg cond$
DECOMPOSE TABLE R INTO S( $\bar{A}, \bar{B}$ ), T( $\bar{A}, \bar{C}$ )	R( $\bar{A}, \bar{B}, \bar{C}$ )	S( $\bar{A}, \bar{B}$ ), T( $\bar{A}, \bar{C}$ )	$R(\bar{x}, \bar{y}, \bar{z}) \rightarrow S(\bar{x}, \bar{y})$ $R(\bar{x}, \bar{y}, \bar{z}) \rightarrow T(\bar{x}, \bar{z})$	$S(\bar{x}, \bar{y}) \rightarrow \exists \bar{z} R(\bar{x}, \bar{y}, \bar{z})$ $T(\bar{x}, \bar{z}) \rightarrow \exists \bar{y} R(\bar{x}, \bar{y}, \bar{z})$
JOIN TABLE R, S INTO T WHERE <i>cond</i>	R( $\bar{A}, \bar{B}$ ), S( $\bar{A}, \bar{C}$ )	T( $\bar{A}, \bar{B}, \bar{C}$ )	$R(\bar{x}, \bar{y}), S(\bar{x}, \bar{z}), cond \rightarrow T(\bar{x}, \bar{y}, \bar{z})$	$T(\bar{x}, \bar{y}, \bar{z}) \rightarrow R(\bar{x}, \bar{y}), S(\bar{x}, \bar{z}), cond$
ADD COLUMN C [AS <i>const</i>   <i>func</i> ( $\bar{A}$ )] INTO R	R( $\bar{A}$ )	R( $\bar{A}, C$ )	$R(\bar{x}) \rightarrow R(\bar{x}, const func(\bar{x}))$	$R(\bar{x}, C) \rightarrow R(\bar{x})$
DROP COLUMN C FROM R	R( $\bar{A}, C$ )	R( $\bar{A}$ )	$R(\bar{x}, z) \rightarrow R(\bar{x})$	$R(\bar{x}) \rightarrow \exists z R(\bar{x}, z)$
RENAME COLUMN B IN R TO C	$R_{V_i}(\bar{A}, B)$	$R_{V_i+1}(\bar{A}, C)$	$R_{V_i}(\bar{x}, y) \rightarrow R_{V_i+1}(\bar{x}, y)$	$R_{V_i+1}(\bar{x}, y) \rightarrow R_{V_i}(\bar{x}, y)$

SQL data migration scripts<sup>3</sup> to evolve the actual schema and to migrate the data. To enable run-time support the system provides two opportunities: (i) a run-time component performing on-line query rewriting, and (ii) a set of SQL views to support the old schema in terms of the new one. To enable this second option in the Deployment phase the system generates composed and optimized (by exploiting integrity constraints) views, output in the form of an SQL script.

A first prototype of the Demo is on-line<sup>4</sup>, together with short video tutorial (i.e, screencast), presenting some of the core system features. A richer and more stable version of the interface is about to be released.

### III. THEORETICAL BACKGROUND

This section briefly summarize the theoretical background of *PRISM*, presented in [8].

#### A. The Operational Mapping

A key concept in *PRISM* is the one of Schema Modification Operators. They represent the main conceptual tool the DB Administrator uses to interact with the system, but also the internal representation of the evolution. In fact, the set of operators presented in Table I is the result of a difficult mediation among conflicting requirements: atomicity, usability, lack of ambiguity, invertibility, and predictability. The operational nature of these operators and the strict usability requirement pushed us to design them by continuously validating their effectiveness and efficiency in modeling real schema evolutions such as the one of: MediaWiki<sup>5</sup>, Joomla<sup>6</sup>, Zen Cart<sup>7</sup>, and TikiWiki<sup>8</sup>. This set of SMOs has already been successfully exploited in [1], [8], [9], [10], and [11].

The SMO syntax is aimed to appeal the public of practitioners by resembling, whenever possible, the DDL portion of the widely accepted standard SQL:2003. Thanks to the precise semantics of each operator the system can easily

verify properties such as redundancy generation, information preservation and invertibility. This is used the key to provide a continuous feedback to the user and thus increase evolution predictability.

#### B. The Logical Mapping

Given the SMOs defined by the user during the first phases of the design methodology, the system automatically derives a representation based on Disjunctive Embedded Dependencies (DEDs)[12] capturing as constraints the relationship between two subsequent schema versions. This translation process, based on the templates shown in Table I, results simple thanks to the atomic, well-defined semantics of each SMO. The result of this translation is a set of DEDs mapping the old and new versions of the schema. The generated DEDs are optimized by composing them. The composition step, see [8] for details, by reducing the overall number of DEDs, increases the performance of the query rewriting process, whose complexity is dominated by the DEDs cardinality.

#### C. The Query Rewriting

The set of DEDs described above, provide a formal mapping between schema versions, which is exploited by the highly-optimized query rewriting engine [6] used in *PRISM* to translate user queries across schema versions. The query rewriting process is based on the chase/backchase procedure discussed in [6]. Given a DED rule  $D$ , a user query  $Q$  containing the left-hand-side of  $D$  is extended by adding the right-hand-side of  $D$  as a conjunct. This step does not modify  $Q$ 's semantics. The same operation is performed by repeatedly considering all the available DEDs until  $Q$  gets to a stable point and is no more changing. The produce query is called the Universal Plan  $U$ . The backchase phase removes atoms from the  $U$  if they can be derived (back) from the remaining one by applying again the chase discussed above. By properly guiding this removal process it is possible to express  $Q$  only in terms of the atoms of a given schema, in our case the current one. The resulting query  $Q'$  will be equivalent to the original query  $Q$ , but expressed on the desired version of the schema. By automating this entire process we achieve, for every information preserving evolution step, evolution transparency, allowing user to issue queries against old schema versions.

<sup>3</sup>The system is capable of generating both MySQL and DB2 compatible SQL.

<sup>4</sup>See <http://yellowstone.cs.ucla.edu/schema-evolution/index.php/PrismDemo>

<sup>5</sup>The software backend of Wikipedia: <http://www.mediawiki.org>

<sup>6</sup>An open-source CMS: <http://www.joomla.org>.

<sup>7</sup>An open-source shopping cart: <http://www.zen-cart.com/>.

<sup>8</sup>An open-source wiki front-end: <http://info.tikiwiki.org/tiki-index.php>.

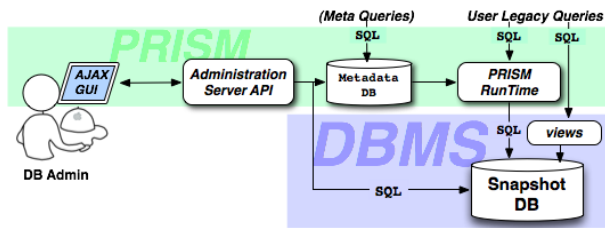


Fig. 3. *PRISM* overall architecture.

#### IV. THE SYSTEM

Figure 3 shows the overall architecture of the system, which can be divided into two main components:

- *administration*: assisting the DBA during the design, and
- *run-time*: transparently supporting legacy queries.

The two components are decoupled by means of the *Metadata DB* which stores all the metadata, i.e., versions of the schema, SMOs and DEDs, into an enhanced *information\_schema*. The *Metadata DB* and its capability of supporting temporal queries on metadata histories have been discussed in [11], while [10] presents a Semantic Web enabled version of it. The DBA is guided by an AJAX web-based interface, which interacts with the *Administration Server API* to design the schema changes. This process is interactive and incremental, at the end of which the evolution is performed on the actual *Snapshot DB*, containing the actual data, and appropriate meta-information are stored in the *Metadata DB*. This portion of the system is only active at design time. At run-time, the corresponding component loads from the *Metadata DB* the schema history and the mappings and, by exploiting the rewriting engine MARS [6], rewrite user queries, expressed on the old schema version against the current one in a transparent way. Alternatively the system (the *Administration Server API*) is capable of generating and installing into the *Snapshot DB* a set of views supporting queries on the old schemas in terms of the data under the current schema version. In this last scenario the role of the run-time portion of the system is played by the DBMS engine itself. The choice between these two options is related to performance issues. The rewriting-based solution allows integrity-constraints based optimizations, but with the cost of on-the-fly rewriting. The view-based approach relies only on the DBMS optimizations, but offers easier integration into existing systems—the run-time is based on the view support of the DBMS in use. The following subsection focuses on the administration component, with particular attention to its supportive interface.

##### A. The Design support

During the design phase the DBA interacts with the AJAX-based Web interface of *PRISM*. The interface is organized in wizard-like fashion, and guides the DBA during the steps of our design methodology, discussed in Section II. While the back-end performance during the design phase were already satisfactory, the asynchronous nature of AJAX has been exploited to further reduce the user waiting times (see [8]), by overlapping user interaction times with the most expensive

back-end tasks. For example, while the user is inserting the list of forward SMOs, the system asynchronously computes the corresponding inverse SMOs and the query support. This is useful especially for complex evolution steps or in the case of a high number of queries. The SMO parsing and analysis has been highly optimized to achieve real-time, on-line checking. The user, while typing, receives continuous feedback on (i) syntax correctness, (ii) information preservation of the inserted operators, (iii) redundancy generation, and (iv) impact of the evolution on the resulting schema. The interface results very responsive and requires limited bandwidth.

##### B. System Validation

The system has been validated on the actual Wikipedia schema evolution history, by exploiting the dataset collected in [1]. We mimicked the schema history, by defining SMO sequences implementing the given evolution. The real Wikipedia queries has been successfully rewritten throughout very long schema histories as shown in [8]. A systematic usability study is part of our research agenda.

#### V. CONCLUSION

*PRISM* represents a major step forward in the support of Schema Evolution, achieving through an intuitive, operational interface a level of *predictability* and *logical independence* of the evolution process not obtained by previous systems. The demonstration highlights the main system functionalities, which will be directly experienced by conference participants. Finally, interesting evolution steps from the schema evolution of popular Web Information Systems will be reviewed in a brief “Saga of Famous Schema Evolutions”.

#### REFERENCES

- [1] C. A. Curino, H. J. Moon, L. Tanca, and C. Zaniolo, “Schema Evolution in Wikipedia: toward a Web Information System Benchmark,” *ICEIS*, To appear: 2008.
- [2] J. Roddick, “A Survey of Schema Versioning Issues for Database Systems,” *Information and Software Technology*, vol. 37, no. 7, pp. 383–393, 1995.
- [3] A. Nash, P. A. Bernstein, and S. Melnik, “Composition of mappings given by embedded dependencies,” in *PODS*, 2005.
- [4] P. A. Bernstein, T. J. Green, S. Melnik, and A. Nash, “Implementing mapping composition,” *VLDB J.*, vol. 17, no. 2, pp. 333–353, 2008.
- [5] R. Fagin, “Inverting schema mappings,” *ACM Trans. Database Syst.*, vol. 32, no. 4, p. 25, 2007.
- [6] A. Deutsch and V. Tannen, “Mars: A system for publishing XML from mixed and redundant storage,” in *VLDB*, 2003.
- [7] L. Chiticariu, M. A. Hernández, P. G. Kolaitis, and L. Popa, “Semi-automatic schema integration in clio,” in *VLDB*, 2007, pp. 1326–1329.
- [8] C. A. Curino, H. J. Moon, and C. Zaniolo, “Graceful database schema evolution: the prism workbench,” in *VLDB*, 2008.
- [9] H. J. Moon, C. A. Curino, A. Deutsch, C.-Y. Hou, and C. Zaniolo, “Managing and querying transaction-time databases under schema evolution,” in *VLDB*, 2008.
- [10] C. A. Curino, L. Tanca, and C. Zaniolo, “Information systems integration and evolution: Ontologies at rescue,” in *Semantic Technologies in System Maintenance (STSM)*, 2008.
- [11] C. A. Curino, H. J. Moon, and C. Zaniolo, “Managing the history of metadata in support for db archiving and schema evolution,” in *Evolution and Change in Data Management (ECDM)*, 2008.
- [12] A. Deutsch and V. Tannen, “Optimization properties for classes of conjunctive regular path queries,” in *DBPL '01: Revised Papers from the 8th International Workshop on Database Programming Languages*. London, UK: Springer-Verlag, 2002, pp. 21–39.