Graceful Database Schema Evolution: the PRISM Workbench

Carlo A. Curino
Politecnico di Milano
carlo.curino@polimi.it

Hyun J. Moon
UCLA
hjmoon@cs.ucla.edu

Carlo Zaniolo
UCLA
zaniolo@cs.ucla.edu

ABSTRACT

Supporting graceful schema evolution represents an unsolved problem for traditional information systems that is further exacerbated in web information systems, such as Wikipedia and public scientific databases: in these projects based on multiparty cooperation the frequency of database schema changes has increased while tolerance for downtimes has nearly disappeared. As of today, schema evolution remains an error-prone and time-consuming undertaking, because the DB Administrator (DBA) lacks the methods and tools needed to manage and automate this endeavor by (i) predicting and evaluating the effects of the proposed schema changes, (ii) rewriting queries and applications to operate on the new schema, and (iii) migrating the database.

Our $\mathcal{PRISM}$ system takes a big first step toward addressing this pressing need by providing: (i) a language of Schema Modification Operators to express concisely complex schema changes, (ii) tools that allow the DBA to evaluate the effects of such changes, (iii) optimized translation of old queries to work on the new schema version, (iv) automatic data migration, and (v) full documentation of intervened changes as needed to support data provenance, database flash back, and historical queries. $\mathcal{PRISM}$ solves these problems by integrating recent theoretical advances on mapping composition and invertibility, into a design that also achieves usability and scalability. Wikipedia and its 170+ schema versions provided an invaluable testbed for validating $\mathcal{PRISM}$ tools and their ability to support legacy queries.

1. INTRODUCTION

The incessant pressure of schema evolution is impacting every database, from the world's largest[1] "World Data Centre for Climate" featuring over 6 petabytes of data, to the smallest single-website DB. DBMSs have long addressed, and largely solved, the physical data independence problem, but their progress toward logical data independence and graceful schema evolution has been painfully slow. Both practitioners and researchers are well aware that schema modifications can: (i) dramatically impact both data and queries [8], endangering the data integrity, (ii) require expensive application maintenance for queries, and (iii) cause unacceptable system downtimes. The problem is particularly serious in Web Information Systems, such as Wikipedia [33], where significant downtimes are not acceptable while a mounting pressure for schema evolution follows from the diverse and complex requirements of its open-source, collaborative software-development environment [8]. The following comment[2] by a senior MediaWiki [32] DB designer, reveals the schema evolution dilemma faced today by DataBase Administrators (DBAs): *"This will require downtime on upgrade, so we're not going to do it until we have a better idea of the cost and can make all necessary changes at once to minimize it."*

Clearly, what our DBA needs is the ability to (i) predict and evaluate the impact of schema changes upon queries and applications using those queries, and (ii) minimize the downtime by replacing, as much as possible, the current manual process with tools and methods to automate the process of database migration and query rewriting. The DBA would also like (iii) all these changes documented automatically for: data provenance, flash-backs to previous schemas, historical queries, and case studies to assist on future problems.

There has been much recent work and progress on theoretical issues relating to schema modifications including mapping composition, mapping invertibility, and query rewriting [21, 14, 25, 4, 13, 12].

These techniques have often been used for heterogenous database integration; in $\mathcal{PRISM}$[3] we exploit them to automate the transition to a new schema on behalf of a DBA. In this setting, the semantic relationship between source and target schema, deriving from the schema evolution, is more crisp and better understood by the DBA than in typical database integration scenarios. Assisting the DBA during the design of schema evolution, $\mathcal{PRISM}$ can thus achieve objectives (i-iii) above by exploiting those theoretical ad-

[1]Source: `http://www.businessintelligencelowdown.com/2007/02/top_10_largest_.html`

[2]From the SVN commit 5552 accessible at: `http://svn.wikimedia.org/viewvc/mediawiki?view=rev&revision=5552`.

[3]$\mathcal{PRISM}$ is an acronym for *Panta Rhei Information & Schema Manager*—'Panta Rhei' (Everything is in flux), is often credited to Heraclitus. The project homepage is: `http://yellowstone.cs.ucla.edu/schema-evolution/index.php/Prism`.

VLDB '08, August 24-30, 2008, Auckland, New Zealand

vances, and prompting further DBA input in those rare situations in which ambiguity remains.

Therefore, $\mathcal{PRISM}$ provides an intuitive, operational interface, used by the DBA to evaluate the effect of a possible evolution steps w.r.t. redundancy, information preservation, and impact on queries. Moreover, $\mathcal{PRISM}$ automates error-prone and time-consuming tasks such as query translation, computation of inverses, and data migration. As a by-product of its use $\mathcal{PRISM}$ creates a complete, unambiguous documentation of the schema evolution history, which is invaluable to support data provenance, database flash backs, historical queries, and user education about standard practices, methods and tools.

$\mathcal{PRISM}$ exploits the concept of Schema Modification Operators (SMO) [4], representing atomic schema changes, which we then modify and enhance by (i) introducing the use of functions for data type and semantic conversions, (ii) providing a mapping to Disjunctive Embedded Dependencies (DEDs), (iii) obtain invertibility results compatible to [13], and (iv) define the translation into efficient SQL primitives to perform the data migration. $\mathcal{PRISM}$ has been designed and refined against several real-life Web Information Systems including MediaWiki [32], Joomla[4], Zen Cart[5], and TikiWiki[6]. The system has been tested and validated against the benchmark for schema evolution defined in [8], which is built over the actual database schema evolution history of Wikipedia (170+ schema versions in 4.5 years). Its ability to handle the very complex evolution of one of the ten most popular website of the World Wide Web[7] offers an important validation of practical soundness and completeness of our approach.

While Web Information Systems represent an extreme case, where the need for evolution is exacerbated [8] by the fast evolving environment in which they operates, every DBMS would benefit from *graceful schema evolution*. In particular every DB accessed by applications inherently "hard to modify" like: public Scientific Databases accessed by applications developed within several independent institutions, DB supporting legacy applications (impossible to modify), and system involving closed-source applications foreseeing high adaptation costs. Transaction time databases with evolving schema represent an interesting scenario were similar techniques can be applied [23].

Contributions. The $\mathcal{PRISM}$ system, harness recent theoretical advances [12, 15] into practical solutions, through an intuitive interface, which masks the complexity of underling tasks, such as logic-based mappings between schema versions, mapping composition, and mapping invertibility. By providing a simple operational interface and speaking commercial DBMS jargon, $\mathcal{PRISM}$ provides a user-friendly, robust bridge to the practitioners' world. System scalability and usability have been addressed and tested against one of the most intense histories of schema evolution available to date: the schema evolution of Wikipedia, featuring in 4.5 years over 170+ documented schema versions and over 700 gygabytes of data [1].

[4]An open-source content management system available at: `http://www.joomla.org`.

[5]A free open-source shopping cart software available at: `http://www.zen-cart.com/`.

[6]An open-source wiki front-end, see: `http://info.tikiwiki.org/tiki-index.php`.

[7]Source: `http://www.alexa.com`.

Paper Organization. The rest of this paper is organized as follows: Section 2 discusses related works, Section 3 introduces a running example and provides a general overview of our approach, Section 4 discusses in details design and invertibility issues of the SMO language we defined, Section 5 presents the data migration and query support features of $\mathcal{PRISM}$. We discuss engineering optimization issues in Section 6, and devote Section 7 to a brief description of the system architecture. Section 8 is dedicated to experimental results. Finally Section 9 and 10 discuss future developments and draw our conclusions.

2. RELATED WORKS

Some of the most relevant approaches to the general problem of schema evolution are the impact-minimizing methodology of [27], the unified approach to application and database evolution of [18], the application-code generation of [7] and the framework for metadata model management of [22] and the further contributions [3, 5, 31, 34]. While these and other interesting attempts provide solid theoretical foundations and interesting methodological approaches, the lack of operational tools for graceful schema evolution observed by Roddick in [29] remains largely unsolved twelve years later. $\mathcal{PRISM}$ represents, at the best of our knowledge, the most advanced attempt in this direction available to date.

The operational answer to the issue of schema evolution used by $\mathcal{PRISM}$ exploits some of the most recent results on mapping composition [25], mapping invertibility [13], and query rewriting [12]. The SMO language used here captures the essence of existing works [4], but extends them with functions, for expressing data type and semantic conversions. The translation between SMOs and Disjunctive Embedded Dependencies (DED) exploited here is similar to the incremental adaptation approach of [31], but achieves different goals. The query rewriting portion of $\mathcal{PRISM}$ exploits theories and tools developed in the context of the MARS project [11, 12]. The theories of mapping composition studied in [21, 14, 25, 4], and the concept of invertibility recently investigated by Fagin et al. in [13, 15] support the notion of SMO composition and inversion.

The big players in the world of commercial DBMSs have been mainly focusing on reducing the downtime when the schema is updated [26] and on assistive design tools [10], and lack the automatic query rewriting features provided in $\mathcal{PRISM}$. Other tools of interest are [20] and LiquiBase[8].

Further related works include the results on mapping information preservation by Barbosa et al. [2], the ontology-based repository of [6], the schema versioning approaches of [19]. XML schema evolution has been addressed in [24] by means of a guideline-driven approach. Object-oriented schema evolution has been investigated in [16]. In the context of data warehouse X-TIME represents an interesting step toward schema versioning by means of the notion of augmenting schema [17, 28]. $\mathcal{PRISM}$ differs form all the above in terms of both goals and techniques.

3. GRACEFUL SCHEMA EVOLUTION

This section is devoted to the problem of schema evolution and to a general overview of our approach. We briefly contrast the current process of schema evolution versus the

[8]Available on-line: `http://www.liquibase.org/`

ideal one and show, by means of a running example, how $\mathcal{PRISM}$ significantly narrows this gap.

Table 1: Schema Evolution: tool support desiderata

Interface	
D1.1	intuitive operational way to express schema changes: well-defined atomic operators;
D1.2	incremental definition of the schema evolution, testing and inspection support for intermediate steps (see D2.1);
D1.3	the schema evolution history is recorded for documentation (querying and visualization);
D1.4	every automatic behavior can be overridden by the user;
Predictability and Guarantees	
D2.1	the system checks for information preservation, and highlights lossy steps, suggesting possible solutions;
D2.2	automatic monitoring of the redundancy generated by each evolution step;
D2.3	impact on queries is precisely evaluated, avoiding confusion over syntactically tricky cases;
D2.4	testing of queries posed against the new schema version on top of the existing data, *before materialization*;
D2.5	performance assessment of the new and old queries, on a (reversible) materialization of the new DB;
Complex Assistive Tasks	
D3.1	given the sequence of forward changes, the system derives an inverse sequence;
D3.2	the system automatically suggests an optimized porting of the queries to the new schema;
D3.3	queries posed against the previous versions of the schema are automatically supported;
D3.4	automatic generation of data migration SQL scripts (both forward and backward);
D3.5	generation and optimization of forward and backward SQL views corresponding to the mapping between versions;
D3.6	the system allows to automatically revert (as far as possible) the evolution step being performed;
D3.7	the system provides a formal logical characterization of the mapping between schema versions;

3.1 Real World

By the current state of the art, the DBA is basically left alone in the process of evolving a DB schema. Based only on his/her expertise, the DBA must figure out how to express the schema changes and the corresponding data migration in SQL—not a trivial matter even for simple evolution steps. Given the available tools, the process is not incremental and there is no system support to check and guarantee information preservation, nor is support provided to predict or test the efficiency of the new layout. Questions, such as "Is the planned data migration information preserving?" and "Will queries run fast enough?", remain unanswered.

Moreover, manual porting of (potentially many) queries is required. Even the simple testing of queries against the new schema can be troublesome: some queries might appear syntactically correct while producing incorrect answers. For instance, all "`SELECT *`" queries might return a different set of columns than what is expected by the application, and evolution sequences inducing double renaming on attributes or tables can lead to queries syntactically compatible with the new schema but semantically incorrect. Schema evolution is thus a critical, time-consuming, and error-prone activity.

3.2 Ideal World

Let us now consider what would happen in an ideal world. Table 1 lists schema evolution desiderata as characteristics of an ideal support tool. We group these features in three classes: (i) *intuitive and supportive interface* which guides the DBA through an assisted, incremental design process; (ii) *predictability and guarantees*: by inspecting evolution steps, schema, queries, and integrity constraints, the system predicts the outcome of the evolution being designed and offers formal guarantees on information preservation, redundancy, and invertibility; (iii) *automatic support for complex tasks*: the system automatically accomplishes tasks such as inverting the evolution steps, generating migration scripts, supporting legacy queries, etc.

The gap between the ideal and the real world is quite wide and the progress toward bridging it has been slow. The contribution of $\mathcal{PRISM}$ is to fill this gap by appropriately combining existing and innovative pieces of technology and solving theoretical and engineering issues. We now introduce a running example that will be used to present our approach to graceful schema evolution.

3.3 Running (real-life) example

This running example is taken from the actual DB schema evolution of MediaWiki [32], a PHP-based software behind over 30,000 wiki-based websites including Wikipedia—the popular collaborative encyclopedia. In particular, we are presenting a simplified version of the evolution step between schema version 41 and 42—SVN[9] commit 6696 and 6710.

```
-- SCHEMA v41 --
old(oid, title, user, minor_edit, text, timestamp)
cur(cid, title, user, minor_edit, text, timestamp,
    is_new, is_redirect)
-- SCHEMA v42 --
page(pid, title, is_new, is_redirect, latest)
revision(rid, pageid, user,  minor_edit, timestamp)
text(tid, text)
```

The fragment of schema shown above represents the tables storing articles and article revisions in Wikipedia. In schema version 41, current and previous revisions of an article have been stored in the separate tables `cur` and `old` respectively. Both tables feature a numeric `id`, the article `title` and the actual `text` content of the page, the `user` responsible for that contribution, the boolean flag `minor_edit` indicating whether the edit performed is of minor entity or not, and the `timestamp` of the last modification.

For the current version of a page additional metadata is maintained: for instance, `is_redirect` records whether the page is a normal page or an alias for another, and `is_new` shows whether the page has been newly introduced or not.

From schema version 42 on, the layout has been significantly changed: table `page` stores article metadata, table `revision` stores metadata of each article revision, and table `text` stores the actual textual content of each revision. To distinguish the current version of each article, the identifier of the most current revision (`rid`) is referenced by the `latest` attribute of `page` relation. The `pageid` attribute of `revision` references to the key of the corresponding `page`. The `tid` attribute of `text` references the column `rid` in `revision`.

These representations seem equivalent in term of information maintained, but two questions arise: *what are the*

[9]See: `http://svn.wikimedia.org/viewvc/mediawiki/trunk/phase3/maintenance/tables.sql`.

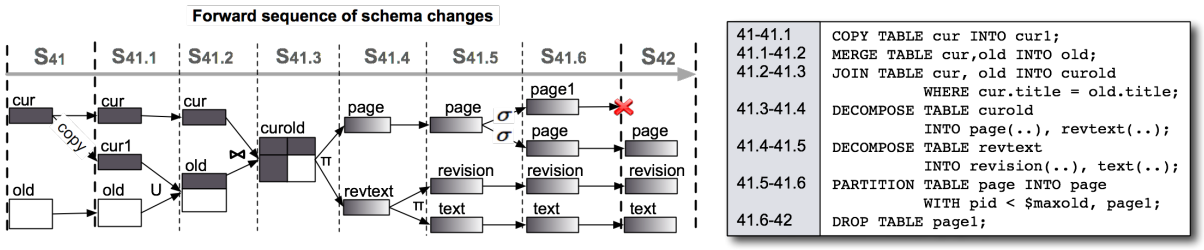

Figure 1: Schema Evolution in Wikipedia: schema versions 41-42

schema changes that lead from schema version 41 to 42? and *how to migrate the actual data?*.

To serve the twofold goal of introducing our Schema Modification Operators (SMO) and answering the above questions, we now illustrate the set of changes required to evolve the schema (and data) from version 41 to version 42, by expressing them in terms of SMOs—a more formal presentation of SMOs is postponed to Section 4.1. Each SMO concisely represents an atomic action performed on both schema *and* data, e.g., MERGE TABLE represents a union of two relations (with same set of columns) into a new one.

Figure 1 presents the sequence of changes[10] leading from schema version 41 to 42 in two formats: on the left, by using the well-known relational algebra notation on an intuitive graph, and on the right by means of our SMO language. Please note that needed, but trivial steps (such as column renaming) have been omitted to simplify the Figure 1.

The key ideas of this evolution are to: (i) make the metadata for the current and old articles uniform, and (ii) regroup such information (columns) into a three-table layout. The first three steps (S_{41} to $S_{41.3}$)—duplication of `cur`, merge with `old`, and join of the merged `old` with `cur`—create a uniform (redundant) super-table `curold` containing all the data and metadata about both current and old articles. Two vertical decompositions ($S_{41.3}$ to $S_{41.5}$) are applied to re-group the columns of `curold` into the three tables: `page`, `revision` and `text`. The last two steps ($S_{41.5}$ to S_{42}) horizontally partition and drop one of the two partitions, removing unneeded redundancy in table `page`.

The described evolution involves only two out of the 24 tables in the input schema (8.3%), but has a dramatic effect on data and queries: more than 70% of the query templates[11] are affected, and thus require maintenance [8].

To illustrate the impact on queries, let us consider an actual query retrieving the current version of the text of a page in version 41:

```
SELECT cur.text FROM cur
WHERE cur.title = 'Auckland';
```

Under schema version 42 the equivalent query looks like:

```
SELECT text.text
FROM page, revision, text
WHERE page.pid = revision.page AND
      revision.rid = text.tid AND
      page.latest = revision.rid AND
      page.title = 'Auckland';
```

[10]While different sets of changes might produce equivalent results, the one presented mimics the actual data migration that have been performed on the Wikipedia data.

[11]The percentage of query instances affected is incredibly higher. Query templates, generated by grouping queries with identical structure, represent an evaluation of the development effort.

3.4 Filling the gap

In a nutshell, $\mathcal{PRISM}$ assists the DBA in the process of designing evolution steps by providing him/her with the concise SMO language used to express schema changes. Each resulting evolution step is then analyzed to guarantee information-preservation, redundancy control and invertibility. The SMO operational representation is translated into a logical one, describing the mapping between schema versions, which enables chase-based query rewriting. The deployment phase consists in the automatic migration of the data by means of SQL scripts and the support of queries posed against the old schema versions by means of either SQL Views or online query rewriting. As a by-product, the system stores and maintains the schema layout history, which is accessible at any moment.

In the following, we describe a typical interaction with the system, presenting the main system functionalities and briefly mentioning the key pieces of technologies exploited. Let us now focus on the evolution of our running example:

Input: a database DB_{41} under schema S_{41}, Q_{old} an optional set of queries typically issued against S_{41}, and Q_{new} an optional set of queries the DBA plans to support with the new schema layout S_{42}.

Output: a new database DB_{42} under schema S_{42} holding the migrated version of DB_{41} and an appropriate support for the queries in Q_{old} (and potentially other queries issued against S_{41}).

Step 1: *Evolution Design*

(i) the DBA expresses, by means of the Schema Modification Operators (SMO), one (or more) atomic changes to be applied to the input schema S_{41}, e.g., the DBA introduces the first three SMOs of Figure 1—*Desiderata: D1.1.*

(ii) the system virtually applies the SMO sequence to input schema and visualizes the candidate output schema, e.g., $S_{41.3}$ in our example—*Desiderata: D1.2.*

(iii) the system verifies whether the evolution is information preserving or not. Information preservation is checked by verifying conditions, we defined for each SMO, on the integrity constraints, e.g., DECOMPOSE TABLE is information preserving if the set of common columns of the two output tables is a (super)key for at least one of them. Thus, in the example the system will inform the user that the MERGE TABLE operator used between version $S_{41.1}$ and $S_{41.2}$ is not

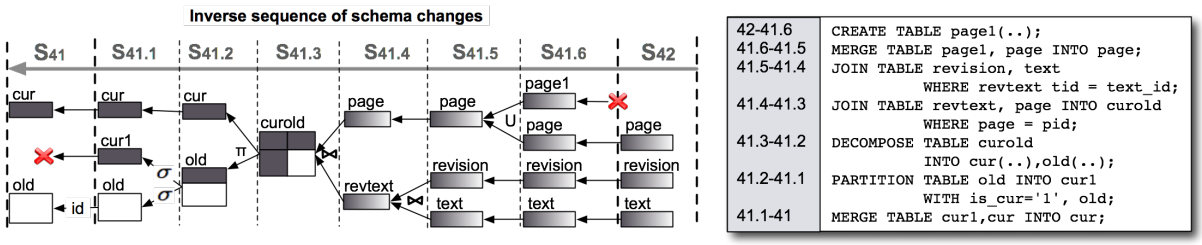

Figure 2: Running example Inverse SMO sequence: 42-41.

information preserving and suggests the introduction of a column `is_old` indicating the provenance of the tuples (discussed in Section 4.2)—*Desiderata: D2.1.*

(iv) each SMO in the sequence is analyzed for redundancy generation, e.g., the system informs the user that the COPY TABLE used in the step S_{41} to $S_{41.1}$ generates redundancy; the user is interrogated on whether such redundancy is intended or not—*Desiderata: D2.2.*

(v) the SMO sequence is translated into a logical mapping between schema versions, which is expressed in terms of Disjunctive Embedded Dependencies (DEDs) [12]—*Desiderata: D3.7.*

The system offers two alternative ways to support what-if scenarios and testing queries in Q_{new} against the data stored in DB_{41}: by means of query rewriting or by means of SQL views.

(vi-a) a DED-based chase engine [12] is exploited to rewrite the queries in Q_{new} into equivalent queries expressed on S_{41}. As an example, consider the following query retrieving the timestamp of the revisions of a specific page:

```
SELECT timestamp FROM page, revision
WHERE pid = page_id AND title = 'Paris'
```

This query is *automatically* rewritten in terms of tables of the schema S_{41} as follows:

```
SELECT timestamp FROM cur
WHERE  title = 'Paris'
 UNION ALL
SELECT timestamp FROM old
WHERE title = 'Paris';
```

The user can thus test the new queries against the old data—*Desiderata: D2.1.*

(vi-b) equivalently the system translates the SMO sequence into corresponding SQL Views $V_{41.3-41}$ to support queries posed on $S_{41.3}$ (or following schema versions) over the data stored in the basic tables of DB_{41}—*Desiderata: D1.2,D3.5.*

(vii) the DBA can iterate Step 1 until the candidate schema is satisfactory, e.g., the DBA introduces the last four SMOs of Figure 1 and obtains the final schema S_{42}—*Desiderata: D1.2.*

Step 2: *Inverse Generation*

(i) the system, based on the forward SMO sequence and the integrity constraints in S_{41}, computes[12] the candidate inverse sequences. Some of the operators have multiple possible inverses, which can be disambiguated by using integrity constraints or interacting with the user. Figure 2 shows the series of inverse SMOs and the equivalent relational algebra graph. As an example, consider the JOIN TABLE operator of the step $S_{41.2}$ and $S_{41.3}$: it is naturally inverted by means of a DECOMPOSE TABLE operator—*Desiderata: D3.1.*

(ii) the system checks whether the inverse SMO sequence is information preserving, similarly to what has been done for the forward sequence. Desiderata: D2.1.

(iii) if both forward and inverse SMO sequences are information preserving, the schema evolution is *guaranteed* to be completely reversible at every stage—*Desiderata: D3.6.*

Step 3: *Validation and Query support*

(i) the inverse SMO sequence is translated into a DED-based logical mapping between S_{42} and S_{41}—*Desiderata: D3.7.*

Symmetrically to what was discussed for the forward case the system has two alternative and equivalent ways to support queries in Q_{old} against the data in DB_{42}: query rewriting and SQL views.

(ii-a) a DED-based chase engine is exploited to rewrite queries in Q_{old} expressed on S_{41} into equivalent queries expressed on S_{42}. The following query, posed on the `old` table of schema S_{41}, retrieves the text of the revisions of a certain page modified by a given user after "2006-01-01":

```
SELECT text FROM old
WHERE title = 'Jeff_V._Merkey' AND
      user = 'Jimbo_Wales' AND
      timestamp > '2006-01-01';
```

It is *automatically* rewritten in terms of tables of the schema S_{42} as follows:

```
SELECT text
FROM page, revision, text
WHERE pid = page AND tid = text_id AND
    latest <> rid AND title = 'Jeff_V._Merkey' AND
    user = 'Jimbo_Wales' AND
    timestamp > '2006-01-01';
```

The user can inspect and review the rewritten queries—*Desiderata: D2.3, D2.4.*

(ii-b) equivalently the system automatically translates the inverse SMO sequence into corresponding SQL Views V_{41-42} supporting the queries in Q_{old} by means of views over the basic tables in S_{42}—*Desiderata: D2.3, D2.4,D3.5.*

(iii) by applying the inverse SMO sequence to schema S_{42}, the system can determine (and show to the user) the portion of the input schema $S'_{41} \subseteq S_{41}$ on which queries are

[12]Some evolution step might not be invertible, e.g., dropping of a column; in this case, the system interacts with the user who either provides a pseudo-inverse, e.g., populate the column with default values, or rollbacks the change, repeating part of Step 1.

Table 2: Schema Modification Operators (SMOs)

SMO Syntax	Input rel.	Output rel.	Forward DEDs	Backward DEDs
CREATE TABLE R($\bar{\mathbf{A}}$)	-	R($\bar{\mathbf{A}}$)	-	-
DROP TABLE R	R($\bar{\mathbf{A}}$)	-	-	-
RENAME TABLE R INTO T	R($\bar{\mathbf{A}}$)	T($\bar{\mathbf{A}}$)	R($\bar{\mathbf{x}}$) → T($\bar{\mathbf{x}}$)	T($\bar{\mathbf{x}}$) → R($\bar{\mathbf{x}}$)
COPY TABLE R INTO T	R_{V_i}($\bar{\mathbf{A}}$)	$R_{V_{i+1}}$($\bar{\mathbf{A}}$), T($\bar{\mathbf{A}}$)	R_{V_i}($\bar{\mathbf{x}}$) → $R_{V_{i+1}}$($\bar{\mathbf{x}}$) R_{V_i}($\bar{\mathbf{x}}$) → T($\bar{\mathbf{x}}$)	$R_{V_{i+1}}$($\bar{\mathbf{x}}$) → R_{V_i}($\bar{\mathbf{x}}$) T($\bar{\mathbf{x}}$) → R_{V_i}($\bar{\mathbf{x}}$)
MERGE TABLE R, S INTO T	R($\bar{\mathbf{A}}$), S($\bar{\mathbf{A}}$)	T($\bar{\mathbf{A}}$)	R($\bar{\mathbf{x}}$) → T($\bar{\mathbf{x}}$); S($\bar{\mathbf{x}}$) → T($\bar{\mathbf{x}}$)	T($\bar{\mathbf{x}}$) → R($\bar{\mathbf{x}}$) ∨ S($\bar{\mathbf{x}}$)
PARTITION TABLE R INTO S WITH *cond*, T	R($\bar{\mathbf{A}}$)	S($\bar{\mathbf{A}}$), T($\bar{\mathbf{A}}$)	R($\bar{\mathbf{x}}$), *cond* → S($\bar{\mathbf{x}}$) R($\bar{\mathbf{x}}$), ¬*cond* → T($\bar{\mathbf{x}}$)	S($\bar{\mathbf{x}}$) → R($\bar{\mathbf{x}}$),*cond* T($\bar{\mathbf{x}}$) → R($\bar{\mathbf{x}}$),¬*cond*
DECOMPOSE TABLE R INTO S($\bar{\mathbf{A}}$,$\bar{\mathbf{B}}$), T($\bar{\mathbf{A}}$,$\bar{\mathbf{C}}$)	R($\bar{\mathbf{A}}$,$\bar{\mathbf{B}}$,$\bar{\mathbf{C}}$)	S($\bar{\mathbf{A}}$,$\bar{\mathbf{B}}$), T($\bar{\mathbf{A}}$,$\bar{\mathbf{C}}$)	R($\bar{\mathbf{x}}$,$\bar{\mathbf{y}}$,$\bar{\mathbf{z}}$) → S($\bar{\mathbf{x}}$,$\bar{\mathbf{y}}$) R($\bar{\mathbf{x}}$,$\bar{\mathbf{y}}$,$\bar{\mathbf{z}}$) → T($\bar{\mathbf{x}}$,$\bar{\mathbf{z}}$)	S($\bar{\mathbf{x}}$,$\bar{\mathbf{y}}$) → ∃$\bar{\mathbf{z}}$ R($\bar{\mathbf{x}}$,$\bar{\mathbf{y}}$,$\bar{\mathbf{z}}$) T($\bar{\mathbf{x}}$,$\bar{\mathbf{z}}$) → ∃$\bar{\mathbf{y}}$ R($\bar{\mathbf{x}}$,$\bar{\mathbf{y}}$,$\bar{\mathbf{z}}$)
JOIN TABLE R, S INTO T WHERE *cond*	R($\bar{\mathbf{A}}$,$\bar{\mathbf{B}}$), S($\bar{\mathbf{A}}$,$\bar{\mathbf{C}}$)	T($\bar{\mathbf{A}}$,$\bar{\mathbf{B}}$,$\bar{\mathbf{C}}$)	R($\bar{\mathbf{x}}$,$\bar{\mathbf{y}}$), S($\bar{\mathbf{x}}$,$\bar{\mathbf{z}}$), *cond* → T($\bar{\mathbf{x}}$,$\bar{\mathbf{y}}$,$\bar{\mathbf{z}}$)	T($\bar{\mathbf{x}}$,$\bar{\mathbf{y}}$,$\bar{\mathbf{z}}$) → R($\bar{\mathbf{x}}$,$\bar{\mathbf{y}}$),S($\bar{\mathbf{x}}$,$\bar{\mathbf{z}}$),*cond*
ADD COLUMN C [AS *const*|*func*($\bar{\mathbf{A}}$)] INTO R	R($\bar{\mathbf{A}}$)	R($\bar{\mathbf{A}}$,C)	R($\bar{\mathbf{x}}$) → R($\bar{\mathbf{x}}$, *const*|*func*($\bar{\mathbf{x}}$))	R($\bar{\mathbf{x}}$,C) → R($\bar{\mathbf{x}}$)
DROP COLUMN C FROM R	R($\bar{\mathbf{A}}$,C)	R($\bar{\mathbf{A}}$)	R($\bar{\mathbf{x}}$,z) → R($\bar{\mathbf{x}}$)	R($\bar{\mathbf{x}}$) → ∃z R($\bar{\mathbf{x}}$,z)
RENAME COLUMN B IN R TO C	R_{V_i}($\bar{\mathbf{A}}$,B)	$R_{V_{i+1}}$($\bar{\mathbf{A}}$,C)	R_{V_i}($\bar{\mathbf{x}}$,y) → $R_{V_{i+1}}$($\bar{\mathbf{x}}$,y)	$R_{V_{i+1}}$($\bar{\mathbf{x}}$,y) → R_{V_i}($\bar{\mathbf{x}}$,y)
NOP	-	-	-	-

supported by means of SMO to DED translation and query rewriting. In our example $S'_{41} = S_{41}$, thus all the queries in Q_{old} can be answered on the data in DB_{42}.

(iv) the DBA, based on this validation phase, can decide to repeat Steps 1 through 3 to improve the designed evolution or to proceed to test query execution performance in Step 4 —*Desiderata: D1.2.*

Step 4: *Materialization and Performance*

(i) the system automatically translates the forward (inverse) SMO sequence into an SQL data migration script[13]—*Desiderata: D3.4.*

(ii) based on the previous step the system materializes DB_{42} differentially from DB_{41} and support queries in Q_{old} by means of views or query rewriting. By default the system preserves an untouched copy of DB_{41} to allow seamless rollback—*Desiderata: D2.5.*

(iii) query in Q_{new} can be tested against the materialized DB_{42} for absolute performance testing—*Desiderata: D2.5.*

(iv) query in Q_{old} can be tested natively against DB_{41} and the performance compared with view-based and query-rewriting-based support of Q_{old} on DB_{42}—*Desiderata: D2.5.*

(v) the user reviews the performance and can either proceed to the final deployment phase or improve performance by modifying the schema layout and/or modify the indexes in S_{42}. In our example the DBA might want to add an index on the `latest` column of `page` to improve the join performance with `revision`—*Desiderata: D1.2.*

Step 5: *Deployment*

(i) DB_{41} is dropped and queries Q_{old} are supported by means of SQL views V_{41-42} or by on-line query rewriting—*Desiderata: D3.3.*

(ii) the evolution step is recorded into an enhanced `information_schema` to allow schema history analysis and schema evolution temporal querying—*Desiderata: D1.3.*

(iv) the system provides the chance to perform a *late rollback* (migrating back all the available data) by generating an inverse data migration script from the inverse SMO sequence—*Desiderata: D3.6.*

Finally desideratum D1.4 and scalability issues are dealt with at interface and system implementation level, Section 7.

[13]The system is capable of generating two versions of this script: a differential one, preserving DB_{41}, and a non-preserving one, which reduces redundancy and storage requirements.

Interesting underlying theoretical and engineering challenges have been faced to allow the development of this system, among which we recall mapping composition and invertibility, scalability and performance issues, automatic translation between SMO, DED and SQL formalisms, which are discussed in details in the following Sections.

4. SMO AND INVERSES

Schema Modification Operators (SMO) represent a key element in our system. This section is devoted to discussing their design and invertibility.

4.1 SMO Design

The set of operators we defined extends the existing proposal [4], by introducing the notion of function to support data type and semantic conversions. Moreover, we provide formal mappings between our SMOs and both the logical framework of Disjunctive Embedded Dependencies (DEDs)[14] and the SQL language, as discussed in Section 5.

SMOs tie together schema and data transformations, and carry enough information to enable automatic query mapping. The set of operators shown in Table 2 is the result of a difficult mediation between conflicting requirements: atomicity, usability, lack of ambiguity, invertibility, and predictability. The design process has been driven by continuous validation against real cases of Web Information System schema evolution, among which we list: MediaWiki, Joomla!, Zen Cart, and TikiWiki.

An SMO is *a function that receives as input a relational schema and the underlying database, and produces as output a (modified) version of the input schema and a migrated version of the database.*

Syntax and semantics of each operator are rather self explanatory; thus, we will focus only on a few, less obvious matters: all table-level SMOs *consume* their input tables, e.g., JOIN TABLE `a,b` INTO `c` creates a new table `c` containing the join of `a` and `b`, which are then dropped; the PARTITION TABLE operator induces a (horizontal) partition of the tuples from the input table—thus, only one condition is specified; NOP represents an identity operator, which performs no action but namespace management—input and output alphabets of each SMO are forced to be disjoint by exploiting the schema versions as namespaces. The use of functions in ADD COLUMN allows us to express in this simple language tasks

[14]DEDs have been firstly introduced in [11].

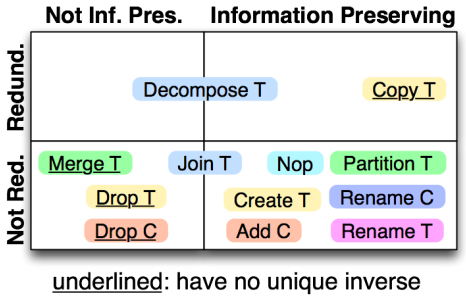

Figure 3: SMOs characterization w.r.t. redundancy, information preservation and inverse uniqueness

such as data type and semantic conversion (e.g., currency or address conversion), and to provide practical ways of recovering information lost during the evolution, as described in Section 4.2.2. The functions allowed are limited to operating at a tuple-level granularity, receiving as input one or more attributes from the tuple on which they operate.

Figure 3 provides a simple characterization of the operators w.r.t. information preservation, uniqueness of the inverse, and redundancy. The selection of the operators has been directed to minimize ambiguity; as a result, only JOIN and DECOMPOSE can be both information preserving and not information preserving. Moreover, simple conditions on integrity constraints and data values are available to effectively disambiguate these cases [30].

When considering sequences of SMOs we notice that: (i) the effect produced by a sequence of SMOs depends on the order; (ii) due to the disjointness of input and output alphabets each SMO acts in isolation on its input to produce its output; (iii) different SMO sequences applied to the same input schema (and data) might produce equivalent schema (and data).

4.2 SMO Invertibility

Fagin et al. [13, 15] recently studied mapping invertibility in the context of source-to-target tuple generating dependencies (s-t tgds) and formalized the notion of quasi-inverse. Intuitively a quasi-inverse is a principled relaxation of the notion of mapping inverse, obtained from it by not differentiating between ground instances (i.e., null-free source instances) that are equivalent for data-exchange purposes. This broader concept of inverse corresponds to the intuitive notion of *"the best you can do to recover ground instances,"* [15] which is well-suited to the practical purposes of $\mathcal{PRISM}$.

In this work, we place ourselves within the elegant theoretical framework of [15] and exploit the notion of quasi-inverse as solid, formal ground to characterize SMO invertibility. Our approach deals with the invertibility within the operational SMO language and not at the logical level of s-t tgds. However, SMOs are translated into a well-behaved fragment of DEDs, as discussed in Section 5. The inverses derived by $\mathcal{PRISM}$, being based on the same notion of quasi-inverse, are consistent with the results shown in [13, 15].

Thanks to the fact that the SMOs in a sequence operate independently, the inverse problem can be tackled by studying the inverse of each operator in isolation. As mentioned above, our operator set has been designed to simplify this task. Table 3 provides a synopsis of the inverses of each

Table 3: SMO inverses

SMO	unique	perfect	Inverse(s)
CREATE TABLE	yes	yes	DROP TABLE
DROP TABLE	no	no	CREATE TABLE COPY TABLE NOP
RENAME TABLE	yes	yes	RENAME TABLE
COPY TABLE	no	no	DROP TABLE MERGE TABLE JOIN TABLE
MERGE TABLE	no	no	PARTITION TABLE COPY TABLE RENAME TABLE
PARTITION TABLE	yes	yes	MERGE TABLE
JOIN TABLE	yes	yes/no	DECOMPOSE TABLE
DECOMPOSE TABLE	yes	yes/no	JOIN TABLE
ADD COLUMN	yes	yes	DROP COLUMN
DROP COLUMN	no	no	ADD COLUMN, NOP
RENAME COLUMN	yes	yes	RENAME COLUMN
NOP	yes	yes	NOP

SMO. The invertibility of each operator can be characterized by considering the existence of a perfect/quasi inverse and uniqueness of the inverse. The problem of uniqueness of the inverse is similar to the one discussed in [13]; in $\mathcal{PRISM}$, we provide a practical workaround based on the interaction with the DBA.

The operators that have a perfect unique inverse are: RENAME COLUMN, RENAME TABLE, PARTITION TABLE NOP, CREATE TABLE, ADD COLUMN, while the remaining operators have one or more quasi-inverses. In particular, JOIN TABLE and DECOMPOSE TABLE represent each other's inverse, in the case of information preserving forward step, and (first-choice) quasi-inverse in case of not information preserving forward step.

COPY TABLE is a redundancy-generating operator for which multiple quasi-inverses are available: DROP TABLE, MERGE TABLE and JOIN TABLE. The choice among them depends on the evolution of the values in the two generated copies. DROP TABLE is appropriate for those cases in which the two output tables are completely redundant, i.e., integrity constraints guarantee total replication. If the two copies evolve independently, and all of the data should semantically participate to the input table, MERGE TABLE represents the ideal inverse. JOIN TABLE is used for those cases in which the input table corresponds to the intersection of the output tables[15]. In our running example the inverse of the COPY COLUMN between S_{41} and $S_{41.1}$ has been disambiguated by the user in favor of DROP TABLE, since all of the data in `cur1` were also available in `cur`.

MERGE TABLE does not have a unique inverse. The three available quasi-inverses differently distribute the tuples from the output table over the input tables. PARTITION TABLE allocates the tuples based on some condition on attribute values; COPY TABLE redundantly copies the data in both input tables; DROP TABLE drops the output table without supporting the queries over the input tables.

DROP TABLE invertibility is more complex. This operator is in fact not information preserving and the default (quasi-)inverse is thus NOP—queries on the old schema insisting on the drop table are thus not supported. However, the user might be able to recover the lost information thanks to redundancy, a possible quasi-inverse is thus COPY TABLE.

[15]Simple column adaptation is also required.

Again in some scenario the drop of a table represents the fact that the table would have been empty, thus a CREATE TABLE will provide proper answers (empty set) to queries on the old version of the schema. These are equivalent quasi-inverses (i.e., equivalent inverses for data-exchange purposes), but, when used for the purpose of query rewriting, they lead to different ways of supporting legacy queries. The system assists the DBA in this choice by showing the effect on queries.

DROP COLUMN shares the same problem as DROP TABLE. Among the available quasi-inverses, there are ADD COLUMN and NOP. The second corresponds to the choice of *not* supporting any query operating on the column being dropped, while the first corresponds to the case in which the lost information can be recovered (by means of functions) from other data in the database. Section 4.2.2 shows an example of information recovery based on the use of functions.

4.2.1 Multiple inverses

$\mathcal{PRISM}$ relies on integrity constraints and user-interaction to select an inverse among various candidates; this practical approach proved effective during our tests.

If the integrity constraints defined on source and target schema do not carry enough information to disambiguate the inverse, two scenarios are considered: the DBA identifies a unique (quasi-)inverse to be used for all the queries, or the DBA decides to manage different queries according to different inverses. In the latter case, typically involving deep constraints changes, the DBA is responsible for instructing the system on how each query should be processed.

As mentioned in Section 3.4, the system always allows the user to override the default system behavior, i.e., the user can specify the desired inverse for every SMO. The user inferface masks most of these technicalities by interacting with the DBA via simple and intuitive questions on the desired effects on queries and data.

4.2.2 Example of a practical workaround

In our running example, the step from $S_{41.1}$ to $S_{41.2}$ merges the tables `cur1` and `old` as follows: MERGE TABLE `cur1`, `old` INTO `old`. The system detects that this SMO has no inverse and assists the DBA in finding the best quasi-inverse. The user might accept a non-query-preserving inverse such as DROP TABLE; however, $\mathcal{PRISM}$ suggests to the user an alternative solution based on the following steps: (i) introduce a column `is_old` in `cur1` and in `old` representing the tuple provenance, (ii) invert the merge operations as PARTITION TABLE, posing a condition on the `is_old` column. This locally solves the issue but introduces a new column `is_old`, which is hard to manage for inserts and updates under schema version 42. For this reason, the user can (iii) insert *after* version $S_{41.3}$ the following SMO: DROP COLUMN `is_old` FROM `curold`. At first, this seems to simply postpone the non-invertibility issue mentioned above. However, the DROP COLUMN operation has, at this point of the evolution, a nice quasi-inverse based on the use of functions:

ADD COLUMN `is_old` AS `strcmp(rid,latest)` INTO `curold`

At this point of the evolution, the proposed function[16] is capable of reconstructing the correct value of `is_old` for each tuple in `curold`. This is possible because the same information is derivable from the equality of the two attributes `latest` and `rid`. This real-life example shows how the system assists the user to create non-trivial, practical workarounds to solve some invertibility issues. This simple improvement of the initial evolution design increases significantly the percentage of supported queries. The evolution step described in our example becomes, indeed, totally query-preserving. Cases manageable in this fashion were more common in our tests than what we expected.

[16]User-defined-functions can be exploited to improve performance.

5. DATA MIGRATION & QUERY SUPPORT

This section discusses $\mathcal{PRISM}$ data migration and query support capabilities, by presenting SMO to DED translation, query rewriting, and SQL generation functionalities.

5.1 SMO to DED translation

In order to exploit the strength of logical languages toward query reformulation, we convert SMOS to the logical language called Disjunctive Embedded Dependencies (DEDs) [11], extending embedded dependencies with disjunction.

Table 2 shows the DEDs for our SMOs. Each SMO produces a forward mapping and backward mapping. Forward mapping tells how to migrate data from the source (old) schema version to the target (new) schema version. As shown in the table, forward mappings do not use any existential quantifier in the right-hand-side, an satisfy the definition of full source-to-target tuple generating dependencies. This is natural in a schema evolution scenario where the mappings are "functional" in that the output database is derived from the input database, without generating new uncontrolled values. The backward mapping is essentially a flipped version of forward mapping, which tells that the target database doesn't contain data other than the ones migrated from the source version. In other words, these two mappings are two-way inclusion dependencies that establish an equivalence between source and target schema versions.

Given an SMO, we also generate identity mappings for unaffected tables between the two versions where the SMO is defined. The reader might be wondering whether this simple translation scheme produces optimal DEDs: the answer is negative, due to the high number of identity DEDs generated. In Section 6.1, we discuss the optimization technique implemented in $\mathcal{PRISM}$.

While invertibility in the general DED framework is a very difficult matter, dealing with invertibility at the SMO level we can provide for each set of forward DEDs (create from our SMO), a corresponding (quasi)inverse.

5.2 Query Rewriting: Chase and Backchase

Using the above generated DEDs, we rewrite queries using a technique called chase and backchase, or C&B [12]. C&B is a query reformulation method that modifies a given query into an equivalent one: given a DED rule D, if the query Q contains the left-hand-side of D, then the right-hand-side of D is added to Q as a conjunct. This does not change Q's answers—if Q satisfies D's left-hand-side, it also satisfies D's right-hand-side. This process is called *chase*. Such query extension is repeated until Q cannot be extended any further. We call the largest query obtained at this point a *universal plan*, U. At this point, the system removes from U every atom that can be obtained back by a chase. This step does not change the answer, either, and it is called *backchase*. U's atoms are repeatedly removed, until no atom can be

dropped any further, whereupon we obtain another equivalent query Q'. By properly guiding this removal phase, it is possible to express Q only by atoms of the target schema.

In our implementation we employ a highly optimized C&B engine called MARS[17] [12]. Using the SMO-generated DEDs and a given query posed on a schema version (e.g., S_{41},) MARS seeks to find an equivalent rewritten query valid on the specified target schema version (e.g., S_{42}.) As an example, consider the query on schema S_{41}:

```
SELECT title, text FROM old;
```

By the C&B process this query is transformed into the following query:

```
SELECT title, text FROM page, revision, text
WHERE pid = pageid AND rid <> latest AND rid = tid
```

This query is guaranteed to produce an equivalent answer but is expressed only in terms of S_{42}.

5.2.1 Integrity constraints to optimize the rewriting

Disjunctive Embedded Dependencies can be used to express both inter-schema mappings and intra-schema integrity constraints. As a consequence, the rewriting engine will exploit both set of constraints to reformulate queries. Integrity constraints are, in fact, exploited by MARS to optimize, whenever possible, the query being rewritten, e.g., by removing semi-joins that are redundant because of foreign keys. The notion of optimality we exploit is the one introduced in [12]. This opportunity further justifies the choice of exploiting a DED-based query rewriting technique.

5.3 SMO to SQL

As mentioned in Section 3.4, one of the key features of $\mathcal{PRISM}$ is the ability to automatically generate data migration SQL scripts and view definitions. This enables a seamless integration with commercial DBMSs. $\mathcal{PRISM}$ is currently operational on MySQL and DB2.

5.3.1 SMO to data migration SQL scripts

Despite their syntactic similarities, SMOs differ from SQL in their inspiration. SMOs are tailored to assist data migration tasks; therefore, many operators combine actions on schema and data, thus providing a concise and unambiguous way to express schema evolution. In order to deploy in relational DBMSs the schema evolution being designed, $\mathcal{PRISM}$ translates the user-defined SMO sequence into appropriate SQL (DDL and DML) statements. The nature of our SMO framework allows us to define, independently for each operator, an optimized sequence of statements implementing the operator semantics in SQL. Due to space limitations, we only report one example of translation. Consider the evolution step $S_{41.1} - S_{41.2}$ of our example:

MERGE TABLE *cur*1,*old* INTO *old*

This is translated into the following SQL (for MySQL):

```
INSERT INTO old
 SELECT cid as oid,title,user,
       minor_edit,text,timestamp
 FROM cur1;
DROP TABLE cur1;
```

[17]See `http://rocinante.ucsd.edu:8080/mars/demo/mars_demo.html` for an on-line demonstration showing the actual chase steps.

While the translation of each operator is optimal when considered in isolation, further optimizations are being considered to improve performance of sequences of SMOs; this is part of our current research.

5.3.2 SMO to SQL Views

The mapping between schema versions can be expressed in terms of views, as it often happens in the data integration field. Views can be used to enable what-if scenarios (forward views,) or to support old schema versions (backward views.) Each SMO can be independently translated into a corresponding set of SQL Views. For each table affected by an SMO, one or more views are generated to virtually support the output schema in terms of views over the input schema (the SMO might be part of an inverse sequence). Consider the following SMO of our running example $S_{41.2} - S_{41.3}$:

JOIN TABLE *cur*, *old* INTO *old* WHERE *cur.title* = *old.title*

This is translated into the following SQL View (for MySQL):

```
CREATE VIEW curold AS
  SELECT * FROM cur,old
  WHERE cur.title = old.title;
```

Moreover, for each unaffected table, an identity view is generated to map between schema versions. This view generation approach is practical only for limited length histories, since it tends to generate long view chains which might cause poor performance. To overcome this limitation an optimization has been implemented in the system. As discussed in Section 6.2, MARS chase/backchase is used to implement view composition. The result consists of the generation of a set of highly optimized, composed views, whose performance is presented in Section 8.

6. SCALABILITY AND OPTIMIZATION

During the development of $\mathcal{PRISM}$, we faced several optimization issues due to the ambitious goal of supporting very long schema evolution histories.

6.1 DED composition

As we discussed in the previous section, DEDs generated from SMO tend to be too numerous for efficient query rewriting. In order to achieve efficiency in query reformulation between two distant schema versions, we compose, where possible, subsequent DEDs.

In general, mapping composition is a difficult problem as previous studies have shown [21, 14, 25, 4]. However, as discussed in Section 5.1, our SMOs produce full s-t tgds for forward mappings, which has been proved to support composition well [14]. We implemented a composition algorithm that is similar to the one introduced in [14], to compose our forward mappings. As explained in Section 5.1, our backward mapping is a flipped version of forward mapping. The backward DEDs are derived by flipping forward DEDs paying attention to: (i) union forward DEDs with the same right-hand-side, and (ii) existentially quantify variables not mentioned in the backward DED left-hand-side.

This is clearly not applicable for general DEDs, but serves the purpose for the simple class of DEDs generated from our SMOs. Since the performance of the rewriting engine are mainly dominated by the cardinality of the input mapping, such composition effectively improves rewriting performance.

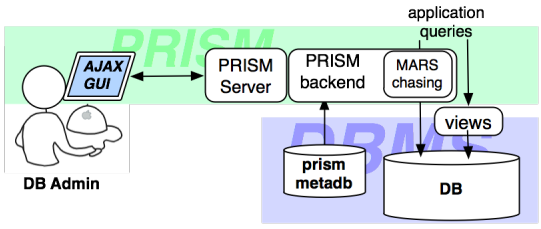

Figure 4: $\mathcal{PRISM}$ system architecture

6.2 View composition

Section 5.3.2 presented the $\mathcal{PRISM}$ capability of translating SMOs into SQL views. This naïve approach has scalability limitations. In fact, after several evolution steps, each query execution may involve long chains of views and thus deliver poor performance. Thanks to the fact that only the actual schema versions are of interest, rather than the intermediate steps, it is possible to compose the views and map the old schema version directly to the most recent one–e.g., in our example we map directly from S_{41} and S_{42}.

View composition is obtained in $\mathcal{PRISM}$ by exploiting the available query rewriting engine. The "body" of each view is generated by rewriting a query representing the "head" of the view in terms of the basic tables of the target schema. For example, the view representing the `old` table in version 41 can be obtained by rewriting the query `SELECT * FROM old` against basic tables under schema version 42. The resulting rewritten query will represent the "body" of the following composed view:

```
CREATE VIEW old AS
 SELECT rid as oid, title, user,
        minor_edit, text, timestamp
 FROM page, revision, text
 WHERE pid = page AND rid = tid AND latest <> rid;
```

Moreover, the rewriting engine can often exploit integrity constraints available in each schema to further optimize the composed views, as discussed in Section 5.2.1.

7. SYSTEM IMPLEMENTATION

$\mathcal{PRISM}$ system architecture decouples an *AJAX* front-end, which ensures a fast, portable and user-friendly interaction from the back-end functionalities implemented in *Java*. Persistency of the schema evolution being designed is obtained by storing intermediate and final information in an extended version of the `information_schema` database, which is capable of storing versioned schemas, queries, SMOs, DEDs, views, migration scripts.

The back-end provides all the features discussed in the paper as library functions invoked by the interface.

The front-end acts as a wizard, guiding the DBA through the steps of Section 3.4. The asynchronous interaction typical of *AJAX* helps to further mask system computation times, this further increase usability by reducing the user waiting times, e.g., during the incremental steps of design of the SMO sequence the system generates and composes DEDs and views for the previous steps.

SMO can also be derived "a posteriori", mimicking a given evolution as we did for the MediaWiki schema evolution history. Furthermore, we are currently investigating automatic approaches for SMO mining from SQL-log integrating $\mathcal{PRISM}$ with the tool-suite of [8].

Table 4: Experimental Setting

Machine	RAM:	4Gb
	CPU (2x):	QuadCore Xeon 1.6Ghz
	Disks:	6x500Gb RAID5
OS	Distribution:	Linux Ubuntu Server 6.06
	Kernel:	2.6.15-26-server
Java	Version:	1.6.0-b105
MySQL	Version:	5.0.22

Queries posed against old schema versions are supported at run-time either by on-line query rewriting performed by the $\mathcal{PRISM}$ backend, which acts in this case as a "magic" driver, or directly by the DBMS in which the views generated at design-time have been installed.

8. EXPERIMENTAL EVALUATION

While in practice it is rather unlikely that a DBA wants to support hundreds of previous schema versions on a production system, we stress-tested $\mathcal{PRISM}$ against an herculean task such as the Wikipedia schema evolution history.

Table 4 describes our experimental environment. The data-set used in these experiments is obtained from the schema evolution benchmark of [8] and consists of actual queries, schemas and data derived from Wikipedia.

8.1 Impact of our system

To assess $\mathcal{PRISM}$ effectiveness in supporting the DBA during schema evolution we use the following two metrics: (i) the percentage of evolution steps fully automated by the system, and (ii) overall percentage of queries supported. To this purpose we select the 66 most common query templates[18] designed to run against version 28 of the Wikipedia schema and execute them against every subsequent schema version[19]. The percentage of schema evolution steps in which the system completely automate the query reformulation activity is: **97.2%**. In the remaining 2.8% of schema evolution steps the DBA must manually rework some of the queries — the following results discuss the proportions of this manual effort. Figure 5 shows the overall percentage of queries automatically supported by the system (74% in the worst case) as compared to the manually rewritten queries (84%) and the original portion of queries that would succeed if left unchanged (only 16%). This illustrates how the system effectively "cures" a wide portion of the failing input queries. The spikes in Figure are due to syntax errors manually introduced (and immediately roll-backed) by the MediaWiki DBAs in the SQL scrips[20] installing the schema in the DBMS, they are considered as outliers in this performance evaluation. The usage of $\mathcal{PRISM}$ would also avoid similar practical issues.

8.2 Run-time performance

Due to privacy issues, the WikiMedia foundation does not release the entire database underlying Wikipedia, e.g, personal user information are not accessible. For this reason, we selected 27 queries out of the 66 initial ones operating on

[18]Each template has been extracted from millions of query instances issued against the Wikipedia back-end database by means of the Wikipedia on-line profiler: `http://noc.wikimedia.org/cgi-bin/report.py?db=enwiki&sort=real&limit=50000`.

[19]Up to version 171, the last version available in our dataset.

[20]As available on the MediaWiki SVN.

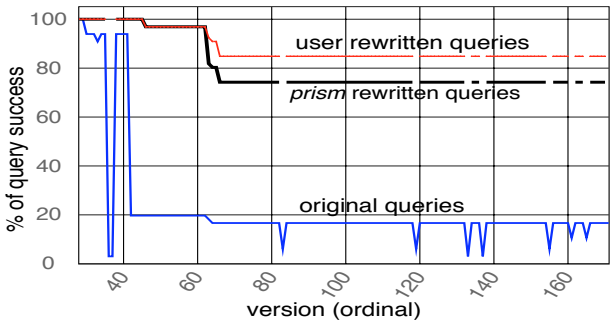

Figure 5: Query success rate on Wikipedia schema versions.

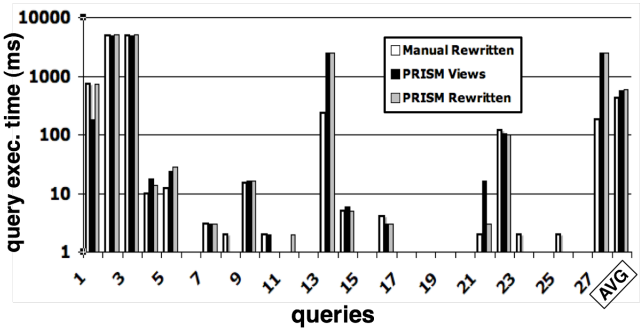

Figure 6: Query execution time on Wikipedia data.

a portion of the schema for which the data were released. The data exploited are a dump, dated *2007-10-07*, of the wiki: "enwikisource"[21]. The database consists of approximately 2,130,000 tuples for about 6.4 Gb of data, capturing the main content of the wiki: articles, revisions, links, images metadata, logs, etc. To show execution times of queries, we selected two key schema versions (28 and 75), which includes the most active portion of the evolution history and correspond to two major releases of the front-end software (1.3 and 1.5.1). Figure 6 shows the execution times of: (i) manually rewritten queries, (ii) original queries executed on top of $\mathcal{PRISM}$-generated views, and (iii) automatically rewritten queries. Since no explicit integrity constraints can be exploited to improve the query rewriting, the two $\mathcal{PRISM}$-based solutions perform almost identically. The 35% gap of performance between manual and automatic rewritten queries is localized in less than 12% of the queries, while the remaining 88% performs almost identically. For such queries, the user was able to remove an unneeded join (between `revision` and `text`) by exploiting his/her knowledge on an implicit integrity constraint (between `old_id` and `rev_id`). The system focuses the DBA's attention on this limited set of under-performing queries (12%) and encourages him/her to improve them manually. Moreover, this overhead is completely removed if we assume this integrity constraint to be fed into $\mathcal{PRISM}$. At the current stage of development, this would require the user to express the schema integrity constraints in terms of DEDs. We plan to provide further support for automatically loading integrity constraints from the schema or inputing them through a simple user interface.

8.3 Usability

Here we focus on response time that represents one of the many factors determining the usability of the system. $\mathcal{PRISM}$ research plans call for a future evaluation of other usability factors as well. To measure user waiting times during design, let us consider the time to compute DED-composition. This is on average 30 seconds for each schema change[22], a reasonable time during the design phase. Part of this time is further masked by the asynchronous nature of our interface. The view generation performance is directly related to query rewriting times. Even without exploiting the DED optimizations discussed in Section 6.1, the view-generation time over the complex evolution step of our running example is on average 980 milliseconds per view, which sums up to 26.5 seconds for the entire set of views, again a reasonable design-time performance. The system is extremely responsive (few milliseconds) when performing operations such as information-preservation analysis, redundancy check, SMO inversion, and SQL script generation.

Finally, it is worthwhile to note that each of the 170 evolution steps of Wikipedia has been successfully modelled by our SMO language. This indicates that $\mathcal{PRISM}$ is practically complete for many applications.

[21]Source: `http://download.wikimedia.org/`.

[22]The composition is incremental and previous step compositions are stored in the enhanced `information_schema`.

9. FUTURE DEVELOPMENTS

$\mathcal{PRISM}$ represents a major first step toward graceful schema evolution. Several extensions of the current prototype are currently being investigated and more are part of our future development plans.

$\mathcal{PRISM}$ exploits integrity constraints to optimize queries and to disambiguate inverses. Currently, our system supports this important feature in a limited and ad-hoc fashion, and further work is needed to fully integrate integrity constraints management into PRISM. Extensions of both the SMO language and the system in this direction are part of our short-term research agenda. Further attention will be devoted to the problem of update management, which will be tackled starting from the solid foundations of $\mathcal{PRISM}$.

As a by-product of the adoption of $\mathcal{PRISM}$, critical information describing the schema evolution history is recorded and currently accessible by the $\mathcal{PRISM}$ interface as discussed in [9].

We plan to support rich temporal queries over this precious collection of metadata and, by integrating $\mathcal{PRISM}$ with the tool-suite of [8], to enable the *a posteriori* analysis of schema evolution histories. This will provide the DBA with a powerful framework that, by generating full documentation of existing evolution histories, is invaluable for: data provenance analysis, database flashback and historical archives.

10. CONCLUSIONS

We presented $\mathcal{PRISM}$, a tool that supports the time-consuming and error-prone activity of Schema Evolution. The system provides the DBA with a concise operational language to represent schema change and increases predictability of the evolution being designed by automatically verifying information preservation, redundancy and query sup-

port. The SMO-based representation of the schema evolution is used to derive logical mappings between schema versions. Legacy queries are thus supported by means of query rewriting or automatically generated SQL views.

The system provides interfaces with commercial relational DBMSs to implement the actual data migration and to deploy views and rewritten queries. As a by-product, the schema evolution history is recorded. This represents an invaluable piece of information for the purposes of documentation, database flash back, and DBA education. Continuous validation against challenging real-life evolution histories, such as the one of Wikipedia, proved invaluable in molding $\mathcal{PRISM}$ into a system that builds on the theoretical foundations laid by recent research and provides a practical solution to the difficult problems of schema evolution.

Acknowledgements

The authors would like to thank Alin Deutsch and Letizia Tanca for the numerous in-depth discussions, MyungWon Ham for the support on the experiments, Julie Nguyen and the reviewers for many suggested improvements. This work was supported in part by NSF-IIS award 0705345: "III-COR: Collaborative Research: Graceful Evolution and Historical Queries in Information Systems—A Unified Approach".

11. REFERENCES

[1] R. B. Almeida, B. Mozafari, and J. Cho. On the evolution of wikipedia. In *Int. Conf. on Weblogs and Social Media*, March 2007.

[2] D. Barbosa, J. Freire, and A. O. Mendelzon. Designing information-preserving mapping schemes for xml. In *VLDB*, pages 109–120, 2005.

[3] P. A. Bernstein. Applying model management to classical meta data problems. In *CIDR*, 2003.

[4] P. A. Bernstein, T. J. Green, S. Melnik, and A. Nash. Implementing mapping composition. *VLDB J.*, 17(2):333–353, 2008.

[5] P. A. Bernstein and E. Rahm. Data warehouse scenarios for model management. In *ER*, 2003.

[6] H. Bounif and R. Pottinger. Schema repository for database schema evolution. *DEXA*, 0:647–651, 2006.

[7] A. Cleve and J.-L. Hainaut. Co-transformations in database applications evolution. *Generative and Transformational Techniques in Software Engineering*, pages 409–421, 2006.

[8] C. A. Curino, H. J. Moon, L. Tanca, and C. Zaniolo. Schema Evolution in Wikipedia: toward a Web Information System Benchmark. *ICEIS*, 2008.

[9] C. A. Curino, H. J. Moon, and C. Zaniolo. Managing the history of metadata in support for db archiving and schema evolution. In *ECDM*, 2008.

[10] DB2 development team. DB2 Change Management Expert. 2006.

[11] 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*, pages 21–39, London, UK, 2002. Springer-Verlag.

[12] A. Deutsch and V. Tannen. Mars: A system for publishing XML from mixed and redundant storage. In *VLDB*, 2003.

[13] R. Fagin. Inverting schema mappings. *ACM Trans. Database Syst.*, 32(4):25, 2007.

[14] R. Fagin, P. G. Kolaitis, L. Popa, and W. C. Tan. Composing schema mappings: Second-order dependencies to the rescue. In *PODS*, pages 83–94, 2004.

[15] R. Fagin, P. G. Kolaitis, L. Popa, and W.-C. Tan. Quasi-inverses of schema mappings. In *PODS '07*, pages 123–132, 2007.

[16] R. d. M. Galante, C. S. dos Santos, N. Edelweiss, and A. F. Moreira. Temporal and versioning model for schema evolution in object-oriented databases. *Data & Knowledge Engineering*, 53(2):99–128, 2005.

[17] M. Golfarelli, J. Lechtenbörger, S. Rizzi, and G. Vossen. Schema versioning in data warehouses. In *ER (Workshops)*, pages 415–428, 2004.

[18] J.-M. Hick and J.-L. Hainaut. Database application evolution: a transformational approach. *Data Knowl. Eng.*, 59(3):534–558, 2006.

[19] H. V. Jagadish, I. S. Mumick, and M. Rabinovich. Scalable versioning in distributed databases with commuting updates. In *Conference on Data Engineering*, pages 520–531, 1997.

[20] T. Lemke and R. Manthey. The schema evolution assistant: Tool description, 1995.

[21] J. Madhavan and A. Y. Halevy. Composing mappings among data sources. In *VLDB*, 2003.

[22] S. Melnik, E. Rahm, and P. A. Bernstein. Rondo: A programming platform for generic model management. In *SIGMOD*, 2003.

[23] H. J. Moon, C. A. Curino, A. D. C.-Y. Hou, and C. Zaniolo. Managing and querying transaction-time databases under schema evolution. In *VLDB*, 2008.

[24] M. M. Moro, S. Malaika, and L. Lim. Preserving XML Queries during Schema Evolution. In *WWW*, pages 1341–1342, 2007.

[25] A. Nash, P. A. Bernstein, and S. Melnik. Composition of mappings given by embedded dependencies. In *PODS*, 2005.

[26] Oracle development team. Oracle database 10g online data reorganization and redefinition. 2005.

[27] Y.-G. Ra. Relational schema evolution for program independency. *Intelligent Information Technology*, pages 273–281, 2005.

[28] S. Rizzi and M. Golfarelli. X-time: Schema versioning and cross-version querying in data warehouses. In *ICDE*, pages 1471–1472, 2007.

[29] J. Roddick. A Survey of Schema Versioning Issues for Database Systems. *Information and Software Technology*, 37(7):383–393, 1995.

[30] J. Ullman. *Principles of Database System*. Computer Science Press, 1982.

[31] Y. Velegrakis, R. J. Miller, and L. Popa. Mapping adaptation under evolving schemas. In *VLDB*, 2003.

[32] Wikimedia Foundation. Mediawiki `http://www.mediawiki.org`, 2007. [Online].

[33] Wikimedia Foundation. Wikipedia `http://en.wikipedia.org/`, 2007. [Online].

[34] C. Yu and L. Popa. Semantic adaptation of schema mappings when schemas evolve. In *VLDB*, 2005.