The VLDB Journal (2013) 22:73–98
DOI 10.1007/s00778-012-0302-x

SPECIAL ISSUE PAPER

Automating the database schema evolution process

Carlo Curino · Hyun Jin Moon · Alin Deutsch · Carlo Zaniolo

Received: 12 January 2012 / Revised: 10 November 2012 / Accepted: 26 November 2012 / Published online: 28 December 2012

Abstract Supporting database schema evolution represents a long-standing challenge of practical and theoretical importance for modern information systems. In this paper, we describe techniques and systems for automating the critical tasks of migrating the database and rewriting the legacy applications. In addition to labor saving, the benefits delivered by these advances are many and include reliable prediction of outcome, minimization of downtime, system-produced documentation, and support for archiving, historical queries, and provenance. The PRISM/PRISM++ system delivers these benefits, by solving the difficult problem of automating the migration of databases and the rewriting of queries and updates. In this paper, we present the PRISM/PRISM++ system and the novel technology that made it possible. In particular, we focus on the difficult and previously unsolved problem of supporting legacy queries and updates under schema and integrity constraints evolution. The PRISM/PRISM++ approach consists in providing the users with a set of SQL-based Schema Modification Operators (SMOs), which describe how the tables in the old schema are modified into those in the new schema. In order to support updates, SMOs are extended with integrity constraints modification operators. By using recent results on schema mapping, the paper (i) characterizes the impact on integrity constraints of structural schema changes, (ii) devises representations that enable the rewriting of updates, and (iii) develop a unified approach for query and update rewriting under constraints. We complement the system with two novel tools: the first automatically collects and provides statistics on schema evolution histories, whereas the second derives equivalent sequences of SMOs from the migration scripts that were used for schema upgrades. These tools were used to produce an extensive testbed containing 15 evolution histories of scientific databases and web information systems, providing over 100 years of aggregate evolution histories and almost 2,000 schema evolution steps.

Keywords Schema evolution · Rewriting · Updates · Mapping · SMO · Integrity constraints management · Relational

C. Curino (✉)
Microsoft, Mountain View, CA, USA
e-mail: ccurino@microsoft.com

H. J. Moon
Google Inc., Mountain View, CA, USA
e-mail: hyunm@google.com

A. Deutsch
UCSD, La Jolla, CA, USA
e-mail: deutsch@cs.ucsd.edu

C. Zaniolo
UCLA, Los Angeles, CA, USA
e-mail: zaniolo@cs.ucla.edu

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 [22], endangering the data integrity, (ii) require expensive application

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

maintenance for queries, and (iii) cause unacceptable system downtimes.

The problem is particularly serious in web information systems, such as Wikipedia [55], 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 [22].

The need for solutions that was already in traditional enterprise environments is made even more pressing by the growing popularity of large web information systems, BigScience projects, and many other open-source, collaborative software development environment [22]. For instance, Wikipedia [55] experienced over 240 schema versions in 6 years, and *Ensembl* Genome [28], a data-intensive, BigScience project, over 410 schema versions in 9 years—we use this as our running example, introduced in Sect. 2.1.

The large number and diversity of stakeholders, and the highly collaborative, fast-progressing environment that is typical of today's enterprise and web and scientific endeavors, in fact accelerate the pace of evolution while reducing tolerance for migration downtime. The urgency of the problem is also supported by our study of the histories of 15 major information systems collected in our schema evolution testbed [19]. This analysis shows the need to provide support for data migration and query adaptation but also integrity constraints and updates: for instance, the *Ensembl* DB schema history contains over 175 individual changes of primary and foreign keys and that almost 20 % of the SQL statements in the Wikipedia workload are legacy updates.

The following comment[2] by a senior MediaWiki [56] DB designer reveals the schema evolution dilemma faced today by database administrators (DBAs): *"This will require downtime on upgrade, so we are 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 and integrity constraint changes upon queries/updates and applications using those queries,
(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,
(iii) document schema evolution automatically thus providing: data provenance, flash-backs to previous schemas, historical queries, and case studies to assist on future problems.

The objective of our research is to provide an integrated solution to these three requirements, making a significant improvement on the current state-of-the art. In fact, the current practice is for the database administrator (DBA) to manually migrate data from the old to the new schema, and for application developers to carefully adapt old applications to operate on the new schema. These operations are extremely time-consuming and error-prone—over 18 % of Wikipedia evolution steps contained errors detectable by an automatic tool.

Until today, this manifest need for schema evolution support remained largely unanswered, even though much progress was made on the related topics, such as mapping composition, invertibility, and query rewriting [26,30,31] that provide the formal basis for sound solutions. The computational costs of these techniques in their general form have prevented their practical deployment, leaving a gulf between the elegant advances in theory and the needs of struggling practitioners—a gulf that has only partially bridged by recent results [16,49].

These techniques have often been used for heterogeneous database integration. The PRISM[3] system we first introduced in [23] exploits such techniques to automate the transition to a new schema on behalf of a DBA. In this setting, the semantic relationship between source and target schema, that is, the schemas before and after evolution, assisting the DBA during the design of schema evolution, PRISM can thus achieve objectives (i–iii) above by exploiting those theoretical advances and prompting further DBA input in those rare situations in which ambiguity remains.

Therefore, PRISM provides an intuitive, operational interface, used by the DBA to evaluate the effect of a possible evolution steps (both structural changes and integrity constraints changes) w.r.t. redundancy, information preservation, and impact on queries and updates. Moreover, 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, PRISM creates automatic documentation of the schema evolution history, which, in our experience, is of great use to support data provenance, database flash-backs, historical queries, and user education about standard practices, methods, and tools.

PRISM exploits the concept of schema modification operators (SMO) [16], 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 [29],

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

[3] 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 as follows: http://yellowstone.cs.ucla.edu/schema-evolution/index.php/Prism.

and (iv) define the translation into efficient SQL primitives to perform the data migration.

In [20], we introduce PRISM++ a significant evolution of our previous attempt. To the best of our knowledge, PRISM++ is *the first system to offer end-to-end support for query and update adaptation through both structural schema changes and integrity constraints evolution*. This is achieved by: (i) complementing the SMO introduced above with a set of integrity constraints modification operators (ICMO), (ii) constraining the use of SMOs, and (iii) extending the basic query rewriting scheme to soundly, yet not completely, supporting query with negation and update rewriting.

Given a specification of the desired schema and integrity constraints evolution, PRISM++ automates the migration of the data and the mapping of (legacy) queries and updates from the old schema to the new schema.

The notion of a sound mapping for queries and updates across schemas has been extensively studied in the past and can be formalized as follows[4]:

Let the current schema be S', the current database instance I', and let S be a past schema version.

1. Given a legacy query Q defined over S, PRISM++ *conceptually* migrates I' "backwards" to an S-instance I, by inverting the schema evolution steps. Then, the result $Q(I)$ of executing Q on I is returned.
2. For a legacy update U against schema S, PRISM++ *conceptually* migrates I' backwards to S-instance I, applies the update to obtain $U(I)$, and then migrates $U(I)$ "forward" through the evolution steps, to obtain a new S'-instance, replacing I'.

The challenge in achieving this semantics is to avoid the prohibitive cost of actually migrating data to support legacy queries or updates. Rather than performing the costly materialization of I, PRISM++ *rewrites* the legacy queries Q and updates U to queries Q' and updates U' *against current schema* S', such that the intended semantics is preserved by operating only on the current database version: $Q'(I') = Q(I)$ and $U'(I')$ is equivalent to executing $U(I)$ and migrating it forward to S'.

The first version of our system, PRISM, achieved these objectives for a large class of queries as described in [23]; however, it did not support updates and could not handle evolution steps that include integrity constraint modifications. In this paper, we focus on the PRISM++ system that overcomes those limitations and supports:

1. *update rewriting* to adapt legacy updates to run on the current schema,
2. *evolution of integrity constraints* significantly extending the class of evolution steps covered, and finally
3. a wider class of queries that now include queries with negation and simple functions.

In addition to these external functionality extensions, major changes were made internally to incorporate the advances made in modeling and mapping legacy update, including

(i) the representation of updates in a fashion that is amenable to rewriting, namely based on query equivalence, (ii) a new inference engine combining novel algorithms and chase-based rewriting technology to rewrite queries and updates through both structural changes of the schema and integrity constraints evolution, and (iii) a set of operators that support modeling of integrity constraint evolution, and a characterization of how integrity constraints are affected by structural schema changes.

In its design, the system balances the need to achieve sufficient expressivity to cover a wide range of practical cases, with computational complexity of several related problems that are notoriously hard in the general case, including the view-update problem [14], deciding schema equivalence [43], schema mapping composition [30] and inversion [31], and consistent query answering [12]. The most general version of the schema evolution problem modeled under these formalisms tends to be intractable or even undecidable (for schema mappings expressed classically, in the language of arbitrary views [53] or of source target tgds [34,38])—see Sect. 9 for a discussion of related work. Thus, the design of PRISM++ uses the evolution language as its main defense against the complexity threat: indeed, this language allows us to "divide and conquer" the tasks, by applying case-by-case analysis for each evolution operator.

Our newly developed testbed [19] provided us with a way to test the expressivity of the PRISM++ evolution language and the effectiveness of our rewriting techniques on the evolution history and workloads (queries and updates) of several real-world systems, including *Ensembl* DB and Wikipedia. A short video demo of PRISM++ is available online.[5]

Furthermore, we introduce two new important companion tools to our system:

– an automatic schema evolution history analysis tool and
– a semi-automatic tool to extract SMO-based evolutions from SQL migration scripts.

The former provides completely automatic data collection and analysis for evolution histories. This tool helps DBAs to asses the intensity and nature of the evolution of their databases by reporting and visualizing several key statistics

[4] This is an adaptation of the classical view-update semantics [14,25,37] to our context, in which evolution operators replace the views.

[5] See: http://tinyurl.com/updaterewriting.

(e.g., lifetime of a table, the number of tables in each revision, frequency of revisioning). This provides a unique vantage point on long and complex evolution histories. We leveraged this tool to scale our data collection and analysis and thus validate several of our hypotheses about frequency and nature of various evolution steps on a broader spectrum of evolution histories—we present this tool early in the paper to motivate the rest of our work by reporting some of our findings.

The latter is a tool that significantly eases the barrier to use our system, by allowing DBAs to write regular SQL to evolve our system (the common practice today), and then feed this into a semi-automatic tool that extracts SMOs. This means that DBAs can leverage the many advantages of our formal framework and rewriting capabilities without the need to immediately embrace a new language. Moreover, the fact that we could build such a tool, and achieve 100 % precision and 66–100 % recall on large evolution histories is a testament of a good design of our set of operators.

Contributions The PRISM++ system harnesses recent theoretical advances [27,31] 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, PRISM++ provides a user-friendly, robust bridge to the practitioners' world.

System scalability and usability have been addressed and tested against some of the most intense histories of schema evolution available to date, including Wikipedia and Ensembl.

PRISM++ is, to the best of our knowledge, the first practical system to address integrity constraints evolution, and update rewriting.

We provide a practical tool for the analysis of schema evolution histories that automates data collection, statistics gathering and visualization for relational database schema versions, and a pattern-based tool that converts SQL migration scripts into sequences of SMOs, thus providing significant support to user aiming at using our tools.

Alltogether PRISM++ is today one of the most complete solutions for schema evolutions we are aware of.

Paper Organization The rest of this paper is organized as follows: Sect. 2 presents a new tool to collect and analyze scheme evolution histories, and introduce our datasets and a running example, Sect. 3 introduces our schema evolution language, Sect. 4 presents the problems related to data migration through SMOs and ICMOs. We then discuss our novel rewriting technology in Sect. 5 and present implementation and optimization concerns in Sect. 6. We then introduce a tool to automatically extract SMOs from SQL scripts in Sect. 7 and evaluate our system and tools in Sect. 8. Related work is discussed in Sect. 9, and we conclude in Sect. 10.

2 Automatic evolution analysis and motivation

In [22], we argued for the need to develop automatic tools for the analysis of long schema evolution histories. This serves two important purposes: in the context of our research, it allows us to automate the collection and analysis of large amount of schema evolution data, to challenge our theories on practical scenarios, and in the context of a database administrator (DBA) everyday life, such a tool provides an useful companion to observe the evolution of a database and detect particularly troubling areas and trends. For example, by analyzing Wikipedia, we detected a clear growing trend that matches both the increase in popularity and increase in features of such a large system. Observing the history the sheer number of changes to tables related to anti-spam features (`ipblocks`, `page_restrictions`, `user_restrictions`, `logging`) highlights the continuous fight against vandalism of the Mediawiki developer team.

To facilitate this information gathering, we built a completely automatic tool that explores both SVN and CVS repositories, downloads multiple versions of a schema, and automatically analyze the evolution, providing several statistics and visualizations. This is a one-stop global view of the evolution for DBAs. We applied this tool to several information systems, some of which are listed in Table 1. A sample of output for the Ensembl database is available at: http://tinyurl.com/evolution-stats.

Among several visualizations that the tool provides, we reproduce Fig. 1, which is the number of Ensembl schema

Table 1 Evolution histories of popular IS in our dataset

System name	System type	# of schema versions	Lifetime (years)
Atutor	Educational CMS	216	7.9
CERN-DQ2	Scientific DB	51	1.3
Coppermine	CMS	36	4.8
Dekiwiki	CRM	16	4.2
E107	CMS	16	8.5
Ensembl	Scientific	412	9.8
KtDMS	CMS	105	6.3
Mediawiki	Wiki	323	8.9
Nucleus	CMS	51	8.9
PHPwiki	CMS	19	7.2
Slashcode	News Website	256	12.5
Tikiwiki	Wiki	152	7.0
TYPO3	CMS	39	8.3
XOOPS	CMS	11	3.5
Zabbix	Monitoring solution	196	10.8

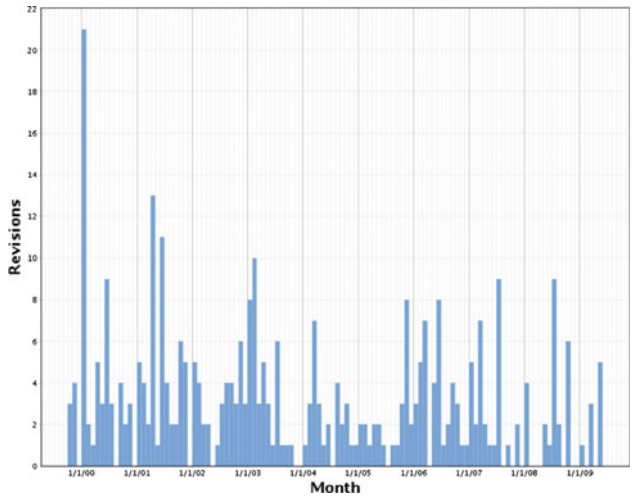

Fig. 1 Sample of our automatic evolution analysis: Ensembl evolution, monthly revision count

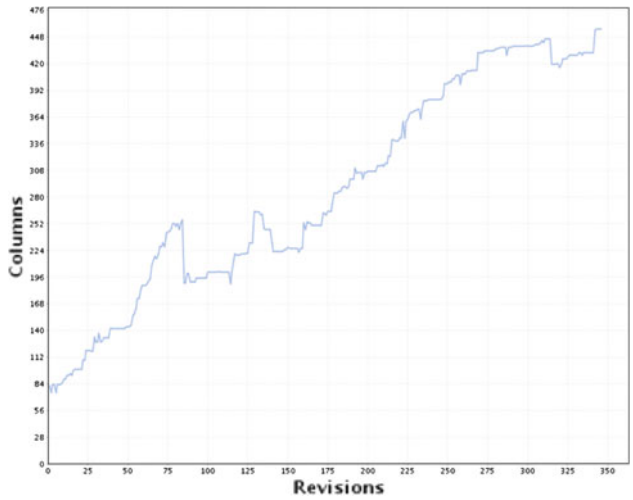

Fig. 2 Sample of our automatic evolution analysis: Ensembl evolution, column count

revisions in each month, and Fig. 2, which shows the number of columns over time. In our experience, this aggregate statistics are very useful to quickly assess the nature of the evolution. As an example of ease of assessment observed in Fig. 2. The constant increase in the number of columns indicates that the database is supporting either a growing set of applications or applications that are quickly augmenting their functionalities, hence requiring new kinds of data to be stored (both effects are at play for Ensembl).

Figure 3 shows another visualization, and it uses the Mediawiki evolution history as an example. This visualization reports aggregated statistics for column evolution, index changes, type changes, rollbacks of the evolution, etc., as well as a combined metric we defined, the "table evolution rate". The table evolution rate has been computed by assigning weights to the various statistics, as a linear regression based

Table Name	Columns evolution	Columns data type changes	Number of roll back	Primary key changes	Foreign key changes	Index changes	Table evolution rate
archive	8	Details...	0	0	0	0	5%
categorylinks	19		0	0	0	1	12%
hitcounter	5		0	0	0	0	2%
image	17	Details...	0	0	0	0	9%
imagelinks	16	Details...	0	0	0	1	10%
interwiki	6	Details...	0	0	0	0	3%
ipblocks	23	Details...	2	0	0	1	31%
logging	29	Details...	1	0	0	0	24%
math	15	Details...	0	0	0	0	8%
objectcache	8	Details...	0	0	0	0	4%
oldimage	31	Details...	2	0	0	0	32%
page	31	Details...	3	0	0	0	40%
pagelinks	14		0	0	0	1	9%

Fig. 3 Sample of our automatic evolution analysis: Mediawiki evolution

on human-generated judgments (by five grad students) of the intensity of the evolution history of a table. The goal of the table evolution rate is to provide a quick estimate/visual clue of how intense the evolution history has been. For example, this metric properly captures the evolution intensity of the `page` table in Mediawiki. The `page` table has been subject to frequent and dramatic refactoring aiming at providing convenient and high performance access to the metadata of an article (the core of the Mediawiki schema).

We argue that this tool allowed us to collect large schema evolution histories (listed in Table 1) and learn a great deal from them. We expect this to be useful for DBAs as well. Most importantly, this tool provided us with a vantage point on schema evolution from which we spotted important limitations of our previous research efforts. Our first work was limited to query rewriting and to structural schema changes only. That was an important first step, but the many evolution histories we collected made evident the need to support updates and integrity constraints evolution as well—this motivates the effort we discuss in this paper.

In the remainder of this section, we introduce a running example that will be used throughout the paper. The running example has been adapted from the actual evolution history of the Ensembl genetic database, one of the richest datasets from Table 1.

2.1 Running example: a genetic DB

The *Ensembl* project[1], funded by the European Biology Institute and the Welcome Trust Sanger Institute, provides a data-centric platform used to support the homonymous human genome database, and other 15 genetic research endeavors. *Ensembl* DB has witnessed an intense schema evolution history. In about *9 years* of life-time over *410+ schema versions* appeared to public (i.e., almost a version a week in the last decade). *Ensembl* users can access the underlying database in multiple ways, including web-page-mediated searches, direct SQL access, and data mining and querying APIs. Every change to the schema potentially impacts all the applications and interfaces built on it; some developed by third parties and therefore hard to upgrade. Hence, there is a substantial need for *transparent* evolution support.

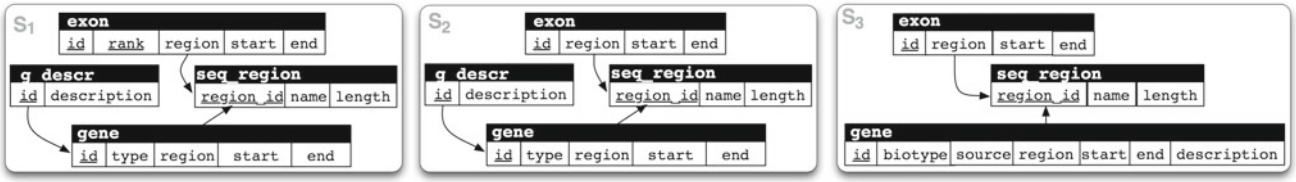

Fig. 4 Three (simplified) schema versions from the actual *Ensembl* genetic DB schema history

We select from this long schema history a few representative examples, compressed and adapted for the sake of presentation. The starting schema S_1 of Fig. 4 is an excerpt of the CVS.[6] schema revision 188.2.6; this schema describes how the *Ensembl* DB stores its information about DNA sequences, exons,[7] and genes. Underlined attributes are primary keys and arrows indicate foreign keys. Each table has a primary key constituted of one numerical identifier, except for the `exon` table, where the rank of an exon is also needed to uniquely identify its tuples. Both `exon` and `gene` refer to *DNA sequence*s stored in table `seq_region`, by referencing their `region_id` and specifying start and end positions in the DNA sequence. The `g_descr` table stores textual descriptions of genes.

In July 2003, the team of DBAs decided to remove from `exon` the `rank` attribute and force `id` to be the new primary key, discarding violating tuples,[8] leading to the schema S_2 in Fig. 4 (revision 188.2.8 CVS schema).

In August 2005, a new evolution step impacting this subset of the schema appeared in the public release of the DB. This evolution step involved two actions: (i) renaming of column `type` to `biotype` in table `gene` and (ii) the joining of the tables `gene` and `g_descr` into a unified table `gene`, leading to the schema S_3 in Fig. 4 (revision 226 of CVS schema). This example is used throughout the paper to illustrate our technical contributions.

Next, we introduce a language to capture schema evolution, which will allow us to divide and conquer the hard problem of query and update rewriting, as we discuss in the rest of the paper.

3 A schema evolution language

3.1 Schema modification operators

In [23], we introduced the SMO of Table 2. SMOs tie together schema and data transformations and carry enough information to enable automatic query mapping. The SMOs in Table 2 are 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.

Table 2 A language for schema evolution: SMO + ICMO

Schema modification operators (SMO) Syntax
`CREATE TABLE R(a,b,c)`
`DROP TABLE R`
`RENAME TABLE R INTO T`
`COPY TABLE R INTO T`
`MERGE TABLE R, S INTO T`
`PARTITION TABLE R INTO S WITH` *cond*`, T`
`DECOMPOSE TABLE R INTO S(a,b), T(a,c)`
`JOIN TABLE R,S INTO T WHERE` *cond*
`ADD COLUMN d [AS` $const \mid func(a,b,c)$`] INTO R`
`DROP COLUMN c FROM R`
`RENAME COLUMN b IN R TO d`
Integrity constraints modification operators (ICMO) syntax
`ALTER TABLE` R `ADD PRIMARY KEY` $pk1(a,b)$`<policy>`
`ALTER TABLE` R `ADD FOREIGN KEY` $fk1(c,d)$ `REFERENCES` $T(a,b)$`<policy>`
`ALTER TABLE` R `ADD VALUE CONSTRAINT` $vc1$ `AS` $R.e =$ "0"`<policy>`
`ALTER TABLE` R `DROP PRIMARY KEY` $pk1$
`ALTER TABLE` R `DROP FOREIGN KEY` $fk1$
`ALTER TABLE` R `DROP VALUE CONSTRAINT` $vc1$

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, for example, `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

[6] See Ensembl CVS repository at: http://tinyurl.com/ensembl-schema.

[7] An *exon* is a nucleic acid sequence related to a portion of DNA.

[8] This information is derived from the CVS logs and from the SQL used for data migration.

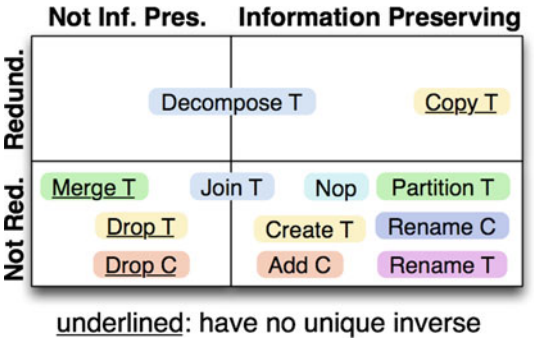

Fig. 5 SMOs characterization w.r.t. redundancy, information preservation, and inverse uniqueness

functions in ADD COLUMN allows us to express in this simple language tasks 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. 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 5 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 [52].

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; and (iii) different SMO sequences applied to the same input schema (and data) might produce equivalent schema (and data).

3.2 Integrity constraint modification operators

Schema modification operators alone do not capture integrity constrains evolution. PRISM++ extends this approach by introducing six new operators used to edit the schema integrity constraints: the ICMOs shown in the second part of Table 2. The "< `policy` >" place-holder is used as a selector to chose among the various integrity constraints enforcement policies offered by PRISM++, as discussed in detail in Sect. 4. PRISM++ supports three basic integrity constraints: *primary keys, foreign keys*, and simple *value constraints.*[9] These constraints cover all the constraints that were actually used in the large dataset of [19]. In the following, we provide the details on how the two sets of operators interact and combine into a powerful and intuitive language for evolution.

Let us start by presenting as an example the evolution step $S_1 - S_2$ of Sect. 2.1. The DBA describes the structural and integrity constraints changes as in the following:

Example 1 Three operators that transform S_1 into S_2

```
1) ALTER TABLE exon DROP PRIMARY KEY pk1;
2) DROP COLUMN rank FROM exon;
3) ALTER TABLE exon ADD PRIMARY KEY pk2(id) ENFORCE;
```

The operators 1 and 3 are ICMOs (introduced by the ALTER keyword), while operator 2 is an SMO.

The keyword `ENFORCE` in the third statement prescribes that the systems will discard all tuples involved in a violation of the newly introduced key. This is only one of the alternative enforcement policies provided by PRISM++, as detailed in Sect. 4.

3.3 Impact of SMO on integrity constraints

Integrity constraint evolution occurs directly (when the administrator adds or removes constraints via ICMOs), for example, adding of a foreign key to the schema, or indirectly (when an SMO changes a schema structure referred to by a constraint), for example, when a table target of a foreign key relationship is joined to another table. An interesting question is thus: "given a set of constraints IC_1 on schema S_1 that is evolved by the sequence of SMOs and ICMOs M into schema S_2, which are the constraints IC_2 that must hold on S_2?"

Formally, we say that IC_2 is *implied by* IC_1 under the evolution M and we denote it as $IC_1 \models_M IC_2$. Note that, for general evolution steps given by arbitrary views and for general classes of integrity constraints, this problem is notoriously hard: checking that a constraint is implied is undecidable, and the implied constraints may have non-finite cover [36].

However, in PRISM++, we do not have to solve the general version of this problem. We only have to deal with three types of supported constraints (key, foreign key, and value) and with simple evolution steps expressed by SMOs and ICMOs—that have been carefully designed to enable all common evolution scenarios while avoiding complexity and decidability pitfalls. It is therefore feasible to pre-compute, for each type of constraint on the initial schema and for each evolution operator, the derived constraints it corresponds to on the evolved schema—we call this *"constraints implication"*.

With reference to Fig. 6, let ic be an integrity constraint for schema S_2, while I_1 and I_2 are instances of S_1 and S_2, respectively, we formally introduce the notion of:

Definition 1 *Constraints implication* Let IC_1 be a set of integrity constraints over schema S_1, and M a mapping from

[9] *These are simple equality assertions about the value of a column and constants, supported by the SQL DDL.*

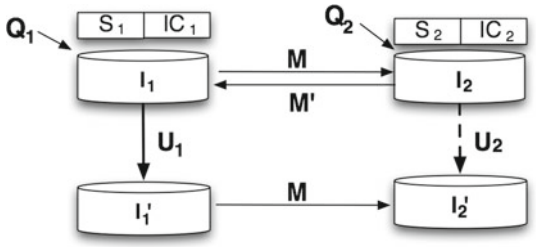

Fig. 6 The general framework

S_1 to S_2, then we write:
$IC_1 \models_M ic$ iff $\forall I_1, I_2 (I_2 = M(I_1) \wedge I_1 \models IC_1 \implies I_2 \models ic)$

Definition 1 says that the integrity constraint ic on schema S_2 is implied by IC_1 under M, if and only if: for every instance I_1 of S_1 and I_2 of S_2 obtained as the mapping of I_1 through M, the following holds: if I_1 satisfies IC_1 then I_2 satisfies ic. [10] The notion of closure is naturally obtained as:

Definition 2 The closure of IC under M is defined: $IC^M := \{ic \mid IC \models_M ic\}$

Thus, IC^M is the set of all integrity constraints implied on S_2 by IC under M. Using this notion of closure, we define the set of all the integrity constraints IC_2 valid on schema S_2 as $IC_2 = IC_1^M$. Applying this definition to each of the structural $SMOs$ defined in Table 2, we obtain a precise characterization of the impact of structural $SMOs$ on integrity constraints.

We exploit the modularity offered by the $SMOs$ to achieve identical results in a programmatic way. In fact, thanks to the independence of the actions performed by each SMO in a sequence, we can derive output constraints observing one SMO at a time (and its input constraints). This reduces the general problem to the one of generating the correct set of output integrity constraints for each SMO type (and each input set of IC), which is easy to achieve in practice, thanks to the atomicity of $SMOs$.

Consider as an example the following input schema S_1 with integrity constraints IC_1:

$S_1 : V(a, b, c)$
$IC_1 : V(a, b, c), V(a, b', c') \implies b = b', c = c'$

And a forward SMO:

```
DECOMPOSE V INTO V1(a,b), V2(a,c)
```

Which transforms schema S_1 into the following schema S_2:

$S_2 : V1(a, b),\ V2(a, c)$

By applying the Definition 2 to IC_1 under the logical mapping M corresponding to the above SMO, we can determine the set of output integrity constraint IC_2 to be the following:

$$\begin{aligned} IC_2 : V1(a, b),\ V1(a, b') &\implies b = b' \\ V2(a, c),\ V2(a, c') &\implies c = c' \\ V1(a, b) &\implies \exists c V2(a, c) \\ V2(a, c) &\implies \exists a V1(a, b) \end{aligned}$$

3.4 Forcing information preservation for SMOs

It turns out that the key technical challenges to PRISM++ data migration and query/update rewriting are raised by those evolution operators that are not information preserving. We introduce the notion of information preservation in terms of the existence of an invertible inverse, intuitively this characterizes a transformation between schemas that can be traversed (by data) back and forth without loss of information.

Definition 3 We say that an evolution operator O from schema S_1 to schema S_2 is *information preserving* if (i) O is functional, that is, for every S_1-instance I_1, there is a unique S_2-instance I_2 with $O(I_1) = I_2$, and (ii) there is an operator O^{-1} from S_2 to S_1 (the *inverse* of O) such that for every S_1-instance I_2, $O^{-1}(O(I_1)) = I_1$.
This notion of information preservation is related to classical notions of invertibility of schema mappings [31], schema equivalence [43], information capacity [42], instantiated to the special case when the schema mapping is given by our evolution operators: O is information preserving if and only if it is invertible, if and only if schemas S_1 and S_2 are equivalent, that is, have the same *information capacity*.

Since non-information-preserving operators require special care, we made the design decision of minimizing their number by normalizing the evolution history so as to force every structural change operator (i.e., every SMO) to apply in a context in which it is information preserving—this is an important difference from [23]. To this end, we successfully exploited ICMOs, which are by definition not information preserving and require special handling anyway (as discussed in Sects. 5.3 and 5.5).

No generality is lost in our approach, since every structural change operator can be sanitized into its information-preserving counterpart by simply adding the proper ICs—whereby any information loss will now be imputed to the sanitizing ICMOs rather than the SMO. This makes the overall set of SMOs and ICMOs a more precise, finer-grained tool for describing evolution—the intuitive advantage is to *separate management of structural modifications from alterations in the information capacity (i.e., IC editing)*.

This is illustrated by Example 2, which displays the operator sequence used to evolve schema S_2 into S_3.

[10] Note that we apply the definition only for the case when M is a functional mapping. This suffices in our context since we force evolution operators to be invertible (as explained below). In general, however, classical schema mappings [34] may associate several possible S_2-instances with a given S_1-instance.

Example 2 Three operators that transform S_2 into S_3

```
1)  RENAME COLUMN type IN gene TO biotype;
2)  ALTER TABLE gene ADD FOREIGN KEY fk2 (id)
       REFERENCES g_descr(id) ENFORCE;
3)  JOIN TABLE gene,g_descr INTO gene
       WHERE gene.id = g_descr.id;
```

The example contains the following evolution steps: (i) renaming of column `type` to `biotype` in table `gene` (operator 1) and (ii) the join of table `gene` and `g_descr` (operator 3), plus the needed integrity constraints change (operator 2).

Operator 2 introduces a foreign key to table `gene`, constraining its values, and thus guaranteeing that the subsequent `JOIN` operator is information preserving (lossless). As one can see, any loss of tuples that would have been incurred by operator 3 is now imputed to operator 2, for example, any tuple in table `gene` not matching a corresponding tuple in `g_descr` is filtered out by operator 2, and the remaining tuples can go through operator 3 with no information loss. Similar sanitizing IC alterations have been studied and identified for each SMO. PRISM++ automatically suggests the sanitizing ICMOs required before each SMO entry and provides feedback on the potential data losses. This is possible because for each SMO, we can statically define a set of pre-conditions under which each the operator is information preserving.

The DBA tightens or relaxes the integrity constraints in the schema, by issuing ICMOs that add or remove such constraints without modifying the schema structure. Issuance of such ICMOs (and the choice of enforcement policies) can: (i) affect the current DB content and (ii) determine the rewriting of queries and updates as discussed in the following. These are the subjects of the next two sections.

4 Data migration

The new evolution language we designed guarantees that data migration steps through SMOs will always be invertible (and information preserving), this significantly simplifies their handling. The focus of this section is thus on migrating data through evolution steps that involve changes of the integrity constraints; in particular, we discuss two policies to handle violations of integrity constraints.

In terms of database content, we will assume that the database satisfies the initial constraints IC_1. Thus, after a constraint is dropped ($IC_2 = IC_1 - k$), the DB instance also satisfies the new relaxed constraints ($I_1 \models IC_2$), and therefore, no additional measures are required. However, when constraints are added ($IC_2 = IC_1 + k$), the original DB instance I_1 might violate the new constraint k and some corrective action is required. PRISM++ helps the DBA in this phase by offering two alternative IC enforcement policies, which are very common in practice [19]. These are `CHECK` and `ENFORCE`.

When `CHECK` is used, the system verifies that the current database satisfies the new constraint k. The ICMO operation is rolled back otherwise. This policy is very common in real-life scenarios, where constraints are often enforced at the application level long before being declared in the schema—for example, all of the foreign keys currently declared in the *Ensembl* genetic DB have been enforced at the application level for years, before being explicitly introduced.

When `ENFORCE` is chosen, the system removes all tuples violating the newly introduced constraint k: if a pair of tuples agrees on the key attributes but disagrees on any non-key attribute, then *both* tuples are removed. If a tuple violates a foreign key constraint, it is removed, and if its removal leads to additional foreign key violations, the removal cascades recursively. The "removed" tuples are not lost: they are stored in the new database instance in temporary *violation tables*, to support any inconsistency resolution action the DBA might wish to carry out. We denote the contents of the violation tables with I_1^{viol}. The remaining tuples form an instance I_1^{sat}, which satisfies the constraint, and which we call the *canonical repair* of I_1.

Our design was motivated by the goal of enabling PRISM++ to work in a permissive mode in which inconsistencies do not halt evolution. PRISM++ supports (but does not mandate) the DBA's intervention for inconsistency resolution. As long as this intervention is delayed (possibly indefinitely), inconsistencies are tolerated and their eventual resolution continues to be supported. Our objective is not to hard-code the "best" repair technique, but to provide the interface in PRISM++ for the DBA or domain expert to plug in their favorite. This is achieved via violation tables and the default repair policy mentioned above. Well-known DB repair techniques (including manual repair) can be applied starting from our canonical repair. Our approach is in contrast to that of *minimal* repairs from the literature [12]. For instance, in a minimal repair, only one of the two tuples violating a key constraint is removed. This suffices to restore consistency and is less invasive. There usually are several minimal repairs possible, and many theoretical works advocate evaluating queries under certain answer semantics over the set of minimal repairs. Minimal repairs are a very attractive concept, but unfortunately, they lead to intractable data complexity of query answering [12] even for very restricted query languages.

In PRISM++, we part from this classical notion and insist on choosing a single repair in order to preserve tractable query answering. This can be the canonical repair that, though non-minimal, is prevalent in practice, and it is also compatible with subsequent conversion into any standard minimal repair (performed by transferring the appropriate tuples back from the violation tables into the instance).

5 A new rewriting technology

In the following, we discuss our rewriting technology. Just as a reminder, PRISM++ implements by means of query/update rewriting the semantics we introduce in the introduction, that is, the rewriting we present is equivalent to a *virtual migration* of the data from the current schema *back* to the past version schema being queried and updated by some legacy application. More precisely, the DBA using PRISM++ evolves the old schema S_1 (with integrity constraints IC_1) into a new schema S_2 (with integrity constraints IC_2) by issuing a sequence M of SMOs and ICMOs. In order to adapt legacy queries and updates designed to work on (S_1,IC_1) to operate on (S_2,IC_2), the system semi-automatically generates an inverse sequence M' conceptually migrating data back. SMOs and ICMOs in the inverse M' determine the semantics of the rewriting.

Extending the rewriting engine, we introduced in [23] to handle integrity constraints, updates and query with negation and functions proved to be a major technical challenge. We first introduce basic query rewriting through SMOs, than discuss the extensions to handle negation and functions, and than present rewriting through ICMOS.

5.1 Query rewriting through SMOs

In order to rewrite queries and updates through SMO-based evolution steps, the PRISM++ system: (i) inverts SMO sequences,[11] (ii) translates each SMO into an equivalent logical schema mapping expressed in the language of DED [27], and (iii) rewrites queries using these DEDs by means of a chase-based algorithm named *chase&backchase (C&B)* [27].

The C&B algorithm reformulates a query on a schema S_1 to an equivalent query on a schema S_2 when the schemas are related by a schema mapping given as a set of DEDs, and when the integrity constraints on the two schemas are expressed as DEDs. DEDs are sufficiently expressive to capture key, foreign key, and all other types of constraints declared in SQL's DDL. This process is an extension of the one discussed in [23], and we only illustrate it via the example in Fig. 7.

Figure 7 shows an example of rewriting through operator 3 of Example 2 (i.e., a `JOIN` SMO). The system automatically inverts the operator by means of a `DECOMPOSE` SMO, and derives a logical mapping between schema versions expressed as DEDs. The DEDs are fed into the C&B rewriting engine [26] to rewrite the input query into an equivalent one operating on the new schema, according to the following semantics:

[11] This process is semi-automatic, and the user is guided by the system in the selection –at evolution time, not at query rewriting time– of the inverse for each SMO [23].

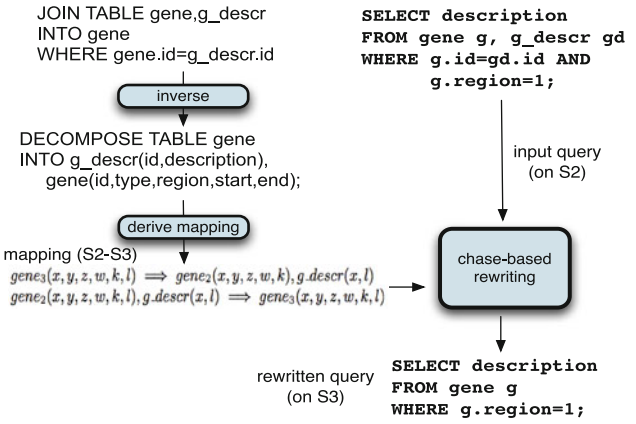

Fig. 7 Query rewriting through SMO

Definition 4 A query Q_2 on schema S_2 is an equivalent rewriting of query Q_1 on S_1 if for every instance I_2 of S_2 the following holds: $Q_2(I_2) = Q_1(M'(I_2))$.

Here, M' is the logical mapping derived from the inverse of the input SMO (e.g., the `DECOMPOSE` SMO of Fig. 7) that conceptually migrates the instance I_2 back to schema S_1. In PRISM++, every SMO step is guaranteed to be information preserving, thus the inverse SMO exists and an M' mapping I_2 to I_1 can easily be derived as in [23].

We can show the following, which extends the results in [23] to incorporate integrity constraints on the schemas:

Theorem 1 *If Q_1 is a union of conjunctive queries, the foreign key constraints on both schemas S_1, S_2 are acyclic, and the SMO operator is information preserving, then an equivalent rewriting Q_2 of Q_1 always exists, and the C&B algorithm of [27] is guaranteed to find one.*

The acyclicity of a set of foreign keys is a classical concept [11], and a special case of the notion of weak acyclicity of a set of embedded dependencies [34]. In essence, it rules out cycles in the dependency graph constructed as follows: the nodes of the graph are the attribute names of all tables in the schema (prefixed by the table name to avoid confusion). For every equality of key attribute K in table R to foreign key attribute F in table S (as asserted by some foreign key constraint), an edge is added from $R.K$ to $S.F$. This acyclicity condition is satisfied by a majority of practical scenarios and widely accepted in the literature as having significant practical impact. Acyclicity (as well as its generalization to weak acyclicity) suffices to guarantee the termination of the chase procedure[11], which the C&B algorithm[27] relies on.

Theorem 1 follows from the facts that (i) the C&B algorithm is guaranteed to terminate when the foreign key constraints are acyclic, (ii) the C&B algorithm is *complete* (i.e. finds a rewriting whenever one exists) for rewriting unions of conjunctive queries across schemas when the schema map-

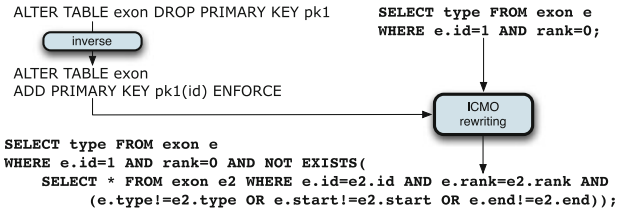

Fig. 8 Query Rewriting through ICMO: ENFORCE

ping is defined by DEDs, and (iii) the sanitized, information-preserving versions of SMOs can be captured using DEDs.

5.2 More expressive query classes

PRISM++ completely automates the rewriting process through mixed sequences of SMOs and ICMOs, by means of a chain of invocations of the chase-based rewriting (for SMOs) or the ICMO rewriting algorithm. The C&B algorithm of [27] was implemented for unions of conjunctive queries (with no negation). For PRISM++, the C&B algorithm is extended to a larger class of queries, which include negation and functions (built-in aggregates and user-defined).

In the example of Fig. 8, we show how negation (e.g., NOT EXISTS) might appear in the rewritten query as a consequence of ICMO-based evolution steps. This introduces a new challenge, since even the chase extensions of [26] cannot deal with this type of negation. To this end, we devised the *QueryRewrite* algorithm that extends the C&B algorithm. Even if the input queries and updates come from the class the C&B can handle, this extension is key to PRISM++, being needed for rewriting queries that contain the type of negation introduced by ICMO rewriting.

The key idea behind the *QueryRewrite* algorithm is to break the input query Q into its *components*, which are maximal query fragments containing no negation or function calls. Each component is rewritten using the standard C&B algorithm, then the rewritten components are re-assembled, preserving the nesting of negation and function calls.

The soundness of *QueryRewrite* derives directly from the fact that we use the chase only on positive (sub)queries without function calls and that the rewritten components are reassembled respecting the structure of the original queries.

Note, however, that this algorithm suffers a loss of completeness that manifests itself as follows: it may be the case that not all of Q's components have an equivalent rewriting in isolation (and therefore none is found by the C&B), yet there is one for the entire query Q, which therefore will be missed by *QueryRewrite*. The loss of completeness is unavoidable due to the undecidability of the rewriting existence problem in the presence of negation and function calls. Nonetheless, the *QueryRewrite* algorithm succeeded in all the practical scenarios from [19] we tested, delivering a significant improvement with respect to the state-of-the art.

This extension is important not only to cover a larger set of queries, but also to handle mixed sequences of SMOs and ICMOs, that, as we will show in the following, can potentially introduce negation in any input query or update.

5.3 Query rewriting in the presence of ICMOs

What we described so far takes care of all structural but information preserving evolution steps. What remain to be discussed is the handling of ICMO-based evolution steps, which in turn involve no structural changes occur. When the DBA *tightens* the set of existing integrity constraints, by introducing a new constraint k, the DBMS will enforce in the new schema S_2 a set $IC_2 = IC_1 + k$ of integrity constraints that implies the old ones, $IC_2 \models_{M'} IC_1$. Old queries and updates can, therefore, be executed *as-is* under IC_2, with no need for any rewriting.[12] Therefore, tightening integrity constraints (difficult for data migration) becomes trivial for rewriting.

On the contrary, relaxing the integrity constraints, for example, issuing an ICMO removing a constraint k, requires a great deal of attention. In fact, queries and updates that assume k is enforced need to be modified to compensate for the lack of such constraint in the new schema. This is formalized by specifying the enforcement policy of the virtual ICMO (the inverse) that re-introduce the removed constraints k—different enforcement policies determine different compensation effects for the missing constraint.

The system provides three policies (selected when specifying the inverse ICMO) that support the most common scenarios found in [19]—they correspond to special cases of the general view-update theory that have great practical appeal:

IGNORE[13]: the system ignores whether the instance I_2 satisfies the integrity constraint k or not. The effect on rewriting is that of executing the original queries and updates *unmodified* on the new schema. Within the view-update literature, this means allowing side effects [14]. While there are clear risks associated with this policy, it must be included to support a very common practice. The system provides appropriate feedback and warnings to the DBA. The subsequent options are stricter and provide stronger guarantees.

CHECK: the rewriting engine checks that the DB instance I_2 satisfies the removed constraint k, for example, in the first step of Example 1 if we apply the CHECK policy, the system would verify that the exon table still satisfies the primary key that has been removed. The original query/update is executed if the condition is evaluated positively and an error

[12] Note that some of the updates will now fail due to the stricter constraints. This is unavoidable to maintain the DB instance I_2 consistent with IC_2 and is in general well-accepted consequence of tightening constraints.

[13] This policy is *only* available for rewriting purposes, that is, for inverses of ICMOs, since the use for data migration would lead to an inconsistent DB instance: $I_2 \not\models IC_2$.

is returned otherwise—these conditions are implemented as probe queries, as shown later in Sect. 5.5 for updates. This policy, as opposed to the previous one, is very conservative and guarantees that queries and updates will operate under the exact same assumptions under which they were designed (i.e., that the constraint k is valid in the DB instance). This is common in scenarios in which the enforcement of some integrity constraints is moved to the application level (e.g., some of the foreign keys in the CERN physics databases [19]). The new applications are designed to enforce the constraint, while the old applications rely on the DBMS for that.

`ENFORCE`: the system introduces conditions in the `WHERE` clause of queries (and updates) to limit the scope of their actions to the canonical repair I_2^{sat} of the DB instance I_2 with respect to the removed constraint—no violating tuples are returned in the query answer (or affected by the update execution). This policy allows the DB instance to partially violate the removed constraint k, limiting the access of legacy queries and updates to the valid portion of the instance (as defined by our canonical, non-minimal repair discussed in Sect. 4). Let us demonstrate this, concentrating on the first operator of Example 1 that *relaxes the primary key* `pk1` *of table* `exon`. The system semi-automatically generates the inverse ICMO that virtually re-introduces the primary key as shown in Fig. 8. The DBA is offered to select the enforcement policy for the inverse ICMO, `ENFORCE` in Fig. 8. The query will be answered on the portion of table `exon` still satisfying the removed primary key $pk1$. This is achieved by introducing an extra condition, that is, the `NOT EXISTS` clause, in the `WHERE` clause to *exclude from the query answer all the tuples violating the primary key*. The algorithm embeds the constraint check in the query. The automatic generation of such conditions is possible given the knowledge of the schema and the constraint being edited and is rather fast—in our implementation takes less than 1ms. This policy has wide applicability in many evolution steps we investigated, in which the old applications operate correctly only when assuming k, while new ones need to violate k.

During the design of the evolution, the DBA, based on his/her understanding of the application needs, selects one of these policies for each inverse ICMO, this gives the DBA completely control on how queries and updates will be rewritten through each evolution step.

5.4 Update rewriting through SMOs

We introduce update rewriting through SMOs by means of the example in Fig. 9, which demonstrates update rewriting through an evolution step decomposing table `exon`.[14] Figure 9 shows that, in order to rewrite SQL updates, the PRISM system: (i) represents the input SQL update in an internal format based on queries, a ÒtrickÓ that is crucial in allowing us to capitalize on query rewriting technology, (ii) rewrites this internal representation through SMO evolution steps, and (iii) converts the rewriting of the internal representation back to a regular SQL update.

The query-based representation of updates completely characterizes the semantics of the update by stating the equivalence of a query posed on the DB instance *before* the update with a query posed on the DB instance *after* the update. Such equivalence describes the relationship between the table contents before and after the update.

The before/after equality of Fig. 9 states that a scan of the table after the execution of the update should produce the same answer of the union of two subqueries posed on the table before the update, returning the tuples not affected by the update as they are, and the tuples being updated with functions/constants in the target list capturing the `SET` action of the update. This kind of representation can be obtained from any SQL update as shown in Table 3.

The rewriting step (ii) transforms this internal representation valid on the old schema, to an equivalent one valid on the new schema, by means of an algorithm we named $UpdateRewrite$.

Algorithm 1: Rewriting algorithm: $UpdateRewrite$

```
Input: U1 ,M' // an update statement on S1 and a schema
       mapping
Output: U2 // the rewritten statement on S2
foreach equivalence R ∈ U1 do
    Rl = left(R);
    Rr = right(R);
    R'l=QueryRewrite(Rl,M');
    R'r=QueryRewrite(Rr,M');
    if R'l = ∅ or R'r = ∅ then
        fail();
    end
    U2 = U2∪(R'l = R'r);
end
```

$UpdateRewrite$ rewrites each query in the equivalence independently, by means of $QueryRewrite$ (the extension handling negation of our query rewriting algorithm, summarized in Sect. 5.1) and produces a similar representation valid under the new schema. Algorithm $UpdateRewrite$ assumes U_1 to be expressed as a set of equivalences between queries on DB instances, and produces an equivalent U_2 in the same format.

The final step (iii) translates this internal representation back to an SQL update statement. This process consists in analyzing the target lists, `FROM` and `WHERE` clauses of the queries and reconstruct the corresponding SQL DDL state-

[14] Note that the evolution is information preserving: (forward) thanks to the primary key on `id`, and (inverse) since the system automatically declares the integrity constraints valid in the output schema (two primary keys on the `id` columns, and two cross foreign keys).

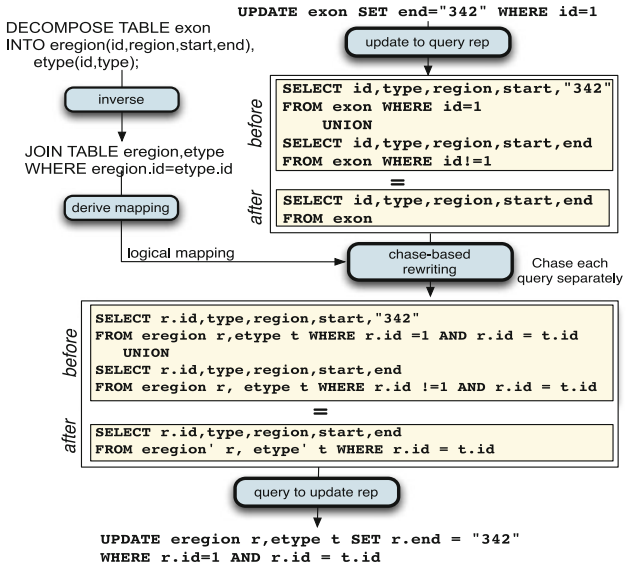

Fig. 9 Update rewriting through SMO

ment(s) valid on the new schema shown in Fig. 9. For insert SQL statement, this operation is trivial, since both representations positively state what should appear in the DB after the execution of the statement, and the translation is purely syntactical. For delete, there is a mismatch between SQL and the query-based representation, where in SQL we specify what to remove, in the mapping-based representation we described the complement, that is, what to keep. Update shares the same issues of delete, where tuples are not removed but modified. Both translations are, therefore, based on inverting the conditions (potentially involving joins with other tables) while propagating the tables to be removed/updated. The system completely automates this process as discussed in Sect. 8, for any insert, update, delete and for any mapping M expressed via SMOs+ICMOs.

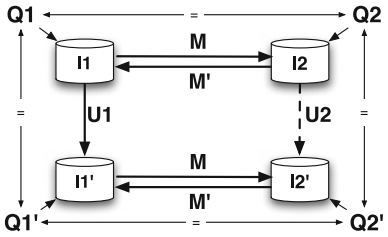

Fig. 10 Update rewriting

The resulting update satisfies the semantics from view-update literature [14,25]:

Definition 5 Let M denote a mapping between schemas S_1 and S_2, with inverse M'. An equivalent rewriting U_2 under schema S_2 (with integrity constraints IC_2) of the original update U_1 under schema S_1 (with integrity constraints IC_1) satisfies the following property: $U_1(M'(I_2)) = M'(U_2(I_2))$

Thanks to the invertibility of both M and M', this leads to a constructive definition of the update on S_2 as follows:

$$U_2 = M(U_1(M^{-1}(I_2))).$$

Based on it, we can make the following claim about algorithm *UpdateRewrite* (in short, we say that *UpdateRewrite* is *sound*):

Theorem 2 *Let M denote a mapping between schemas S_1 and S_2, with inverse M'. Then, for every update U_1 under schema S_1, a successful execution of UpdateRewrite on U_1, M and M' produces an update U_2 under S_2 such that: $U_2 = M \circ U_1 \circ M'$.*

Proof of Theorem 2 We refer to Fig. 10. We start from an update U_1, defined in terms of queries Q_1 and Q_1' as follows: $U_1(I) = I' \iff Q_1(I) = Q_1'(I')$.

Table 3 Query-equivalence-based representation of updates

SQL statement	Query before the update		Query after
INSERT INTO exon VALUES(1,2,3,4,5)	SELECT "1","2","3","4","5" UNION SELECT id,type,region,start,end FROM exon	=	SELECT * FROM exon
INSERT INTO exon (SELECT a,b,c,d,e FROM some_table)	SELECT a,b,c,d,e FROM some_table UNION SELECT id,type,region,start,end FROM exon	=	SELECT * FROM exon
DELETE FROM exon WHERE id =1	SELECT id,type,region,start,end FROM exon WHERE id !=1 SELECT id,type,region,start,"342"	=	SELECT * FROM exon
UPDATE exon SET end="342" WHERE id =1	FROM exon WHERE id =1 UNION SELECT id,type,region,start,end FROM exon WHERE id !=1	=	SELECT * FROM exon

Let Q_2, Q_2' be the rewritings of Q_1, Q_1' via the C&B algorithm. We want to show that the update U_2, defined by

$$U_2(I_2) = I_2' \iff Q_2(I_2) = Q_2'(I_2')$$

is an equivalent rewriting of U_1, that is,

$$U_2(I_2) = M(U_1(M'(I_2))).$$

Denote $U_2(I_2) = I_2'$, $I_1 = M'(I_2)$, and $I_1' = M'(I_2')$. Since M' is the inverse of M, we have $I_2 = M(I_1)$, and $I_2' = M(I_1')$.

From Theorem 1, we know that the C&B yields equivalent rewritings, that is, $Q_2(I_2) = Q_1(M'(I_2))$, and $Q_2'(I_2') = Q_1'(M'(I_2'))$.

Let

$$U_2(I_2) = I_2'.$$

By definition of U_2, this yields

$$Q_2(I_2) = Q_2'(I_2')$$

which by Theorem 1 gives

$$Q_1(M'(I_2)) = Q_1'(M'(I_2'))$$

which, by notation, is equivalent to

$$Q_1(I_1) = Q_1'(I_1')$$

which in turn, by definition of U_1, holds iff

$$U_1(I_1) = I_1'. \tag{1}$$

This immediately implies our claim, since

$$U_2(I_2) = I_2' = M(I_1') \overset{(1)}{=} M(U_1(I_1)) = M(U_1(M'(I_2))).$$

□

5.5 Update rewriting through ICMOs

Once again, tightening of integrity constraints is not challenging for rewriting (since the DBMS enforces a stricter set of constraints $IC_2 = IC_1 + k$) while relaxing integrity constraints requires attention—legacy updates need to be rewritten to operate on a database for which the DBMS only enforces less restrictive integrity constraints ($IC_2 = IC_1 - k$). Update rewriting through ICMOs is similar to the rewriting of queries described early in this section. The key difference is that on top of the conditions checked for queries, updates require *extra* conditions to verify the compliance of the DB instance with the (old) constraints *after* the statement is executed. In the following, we refer to I_2^{viol} as the portion of the DB instance I_2 that violates the (dropped) constraint k.

We discuss here only the extra conditions introduced for updates for each enforcement policy:

IGNORE: no checks are performed, and the update statement is executed *as-is* on the new schema, that is, I_2^{viol} might be not empty and might be affected by the update. This implies potential side effects, and the semantic of update execution is not the original one. Intuitively, this represents the "natural" extension of the update effect on the new schema. The DBA is warned and instructed by the system interface on the effect of this policy. This scenario is common in practice, where changes to the integrity constraints are not reflected into changes to updates, and is thus a must-have in our system.

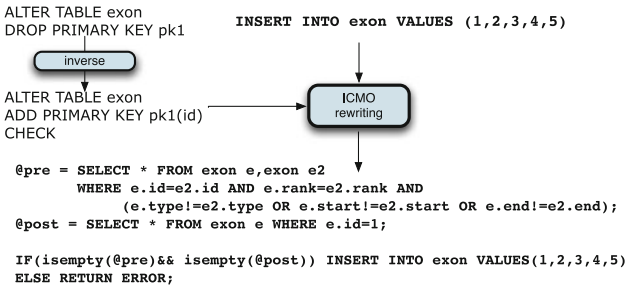

Fig. 11 Update rewriting through ICMO: CHECK

CHECK: PRISM++ checks that the constraint k is satisfied by the DB instance also *after* the update execution, that is, $U_2(I_2) \models k$. This is done by issuing queries before the update execution that check both conditions, and executing the update only if both are satisfied. As an example, consider Fig. 11, where we rewrite an INSERT statement through the same evolution of Fig. 8, but with the CHECK policy for the inverse ICMO. The system checks pre- and post-conditions, automatically derived by analyzing the input statement, to guarantee that the content of table exon respects the primary key, both before and after the execution of the update.

ENFORCE: The system checks that the set of tuples violating the constraints is not change by the execution of the update. This check is performed issuing Boolean queries generated by analyzing the input statement, in a fashion similar to what was discussed above for CHECK. The formal requirement verified by the system is that:

$$I_2^{viol} = U_2(I_2)^{viol}.$$

Next, we discuss some implementation and optimization concerns.

6 Implementation and optimization

The PRISM++ system has been implemented in Java and is loosely based on our prior system [23], but the rewriting engine has been completely redesigned to handle updates, integrity constraints, and queries with negation and functions. The rewriting time performance of our system is a critical metric for success in practical scenarios. Significant effort has been devoted to speeding up the rewriting time for updates, and for schema containing many foreign keys.

PRISM++ computes the rewriting of queries and updates by applying the combination of algorithms described in this

paper. While the newly introduced rewriting through ICMOs is really fast, the rewriting through SMOs of both queries and updates relies on the *chase* procedure [26] that even in the very optimized implementation we use [27] is intrinsically expensive. Furthermore, PRISM++ makes a more intense use of the chase, since each query might be transformed into multiple queries in order to handle negation, and since the explicit manipulation of integrity constraints requires us to increase the size of the mapping managed by the chase engine.

The execution time of the chase is dominated by the size of its input, which includes the integrity constraints from each schema version and a logical mapping between schemas that PRISM++ derives automatically from our operators. The two main avenues to achieve performance are the following: (i) speeding up the rewriting by containing the size of the chase input and (ii) caching previously rewritten queries/updates, thus amortizing the rewriting cost.

Speeding up rewriting To speed up rewriting, we exploit the following key optimizations: (i) an adaptation of the mapping compression approach of [23] (exploiting composition to reduce the size of the chase input), (ii) a mapping pruning technique, extending the basic principle sketched in [44], that removes from the input to the chase mappings and integrity constraints not relevant for the rewriting of a given query/update.

The first compression works by composing long chains of logical mappings into a single mapping connecting directly distant schema versions. This reduces the size of the input of the C&B procedure leading to a significant speed up. Since this technique was present in the prior literature, we use it as our baseline in the experiments in Sect. 8.1.

The second technique is obtained as an extension of the pruning approach discussed in [44]. We refer to the query footprint as the portion of the schema required to answer the query. Pruning operates by analyzing the input query footprint and removing from the input of the C&B procedure all the logical mappings that are not necessary for the rewriting (i.e., predicates about portions of the schema not included in the query footprint). In addition, pruning removes all the schema versions from a schema history not required (e.g., prior to the schema version used in the query). The optimization technique implemented in PRISM++ is a significant extension of the one sketched in [44]. Our implementation can, in fact, also operate in the presence of foreign keys (i.e., by extending the notion of query footprint to all the tables directly or indirectly reachable via foreign keys from the initial footprint) and can manage update statements, by extending the analysis component to deal with update syntax. It is thus presented as an optimization in the experiments in Sect. 8.1.

Furthermore, the actual implementation of the algorithms presented here has been subject to further optimization. In fact, some of the queries produced by the translation steps (to represent an update) have identical portions. Whenever possible, we avoid invocations to the C&B rewriting procedure by reusing results produced for similar queries (this is also part of our baseline performance). A more general-purpose caching technique is presented next.

Caching Observing the workloads from Wikipedia, *Ensembl* and the other information systems from Table 1, we noticed that it is very common for the workload of a system to be based on a rather limited number of query/update templates, which are parametrized and reused multiple times (this is natural, since most queries are issued by applications, in which they are hard-coded as prepared SQL statements). PRISM++ exploits this fact by employing a caching strategy implemented as follows: (i) given an input statement (query or update), PRISM++ extracts a template (by parametrizing it, as for prepared SQL statements), (ii) look-up in an hash-map structure for a matching input template, (iii) retrieve the *rewritten* template if available, and (iv-a) substitute the parameters with the original input values. In case of a cache miss (iv-b), the query/update is rewritten and the system extracts a template from the rewritten query/update and stores it in the cache for later use. Testing with the Wikipedia workload, we also noticed that many templates we extracted only differed in the name of the DB they were targeting (Wikipedia has many DB sharing an identical schema). To this purpose, we adapted the template extraction to be able to cache templates across multiple DBs sharing the same schema. This simple feature (that can be turned on or off) proved very effective in the case of Wikipedia, almost doubling the effectiveness of the cache, as demonstrated in the experiments in Sect. 8.1.

7 Automating the extraction of SMO

We devote this section to introducing a semi-automatic approach for extracting SMOs from SQL-based migration scripts. The tool has been built to ingest the MySQL dialect of SQL, since most of our dataset examples come from open-source projects targeting MySQL as backend DBMS. Extension to other SQL dialects is not conceptually hard, but represents an engineering challenge outside the scope of this effort.

The approach and tools described here have been evaluated on a large testbed of schema evolution histories [19], with encouraging results since the vast majority of SMOs were automatically derived using the methods and tools described next. This suggests that the set of SMOs we present can capture typical schema evolution scenarios in a natural fashion, in addition to achieving important formal properties, including invertibility properties, and correspondence to DED-based mappings. Thus, PRISM/PRISM++'s automatic approach delivers many of the benefits of formal schema mappings while tackling the usability barrier that stands in

the way to their practical adoption in support of the schema evolution process.

It turns out that, from an extraction standpoint, structural schema changes (i.e., the one captured by SMOs) are the most challenging to handle, since they are potentially represented by complex chains of SQL statements. Therefore, our description here focuses on SMOs; ICMO extraction is part of an ongoing development effort, not discussed here.

The approach we present takes as input two subsequent schema versions S_1 and S_2, and a SQL script P, that is used to "patch" schema S_1 into schema S_2. The SQL script P describes a series of steps that change the schema and migrate the data across the two schemas. The amount of information available here is strictly larger than what is typically available in schema mapping scenarios: we know that every table/column that is not mentioned in the migration script P is an identity mapping across schemas S_1 and S_2. Furthermore, the patch P captures in an operational form the entire knowledge of how schema S_1 and S_2 relate. The goal of our tool is to extract such knowledge from the patch P, by deriving a corresponding series of SMOs.

Note that it is reasonable to expect the existence of such SQL migration script, since developers routinely provide upgrade mechanisms for their applications, and such mechanisms for the data management subsystem are typically based on SQL scripts—for example, all of the datasets in our benchmark share this characteristic. Furthermore, it is worth noting that the vast majority of the migration scripts we encountered in practice operate an in-place migration, that is, they patch the existing database in situ. An alternative scenario is the one in which a new DB schema is created and data are copied into it. Our guess is that the in-place procedure is preferred in practice for performance reasons, since all unaffected tables do not impose extra data movements (key for massive databases); however, other operational concerns could come into play favoring the copy-based approach. While we focus our discussion on the in-place form of evolution, our methods naturally apply to the copy-based scenario simply observing that for all the tables not modified by the evolution step under consideration, the data transfer from the old database to the new one is a simple identity mapping, akin to a table rename where the schema name is used as a prefix for the table name, or to a copy pattern as discussed below.

Many of the evolution steps described in the SQL script P (e.g., add a column, create a table) are simple to recognize and translate into corresponding SMOs. However, other transformations, including joins, decompositions, partitions, merges, and table copies, are considerably more complex, and they are therefore processed in three steps, as follows:

1. The SQL statements in P are clustered into independent groups. This is done by building a dependency graph among statements and computing its connected components.
2. We attempt to match each group of statements produced in Step 1 with a library of patterns. (Each SMO might be expressed by more than one pattern of SQL statements.)
3. If (2) fails, we attempt to match the patterns in our library with every subset of statements in the groups from (1).

Step 1 is required because a typical patch P contains multiple schema changes (e.g., various tables are independently evolved). Step 2 exploits a practical observation: the vast majority of "complex" modifications (e.g., joins, decompositions, merges, partitions) are typically captured by sequences of SQL statements, where each statement depends on the previous one. Step 3, while potentially computationally expensive, is required to handle cases in which an evolution steps contains multiple SMOs operating on a common set of tables, in particular when the corresponding SQL statements are intermingled. A common scenario is the one in which simple renamings or additions of columns are intertwined with complex SMOs such as table decomposition or join. Fortunately, the sizes of connected components in our graph were very small (and we expect this to hold true in many practical evolution scenarios); furthermore, simple heuristics were applied to limit the number of subsets considered (e.g., no subsets with more than K entries if K is the largest pattern we consider). This allowed us to handle this exponential step without significant problems. In the rest of this section, we provide further details on our system and the SQL2SMO translation tool that is at its core.

7.1 Dependency graph

The idea behind the dependency graph is to represent the dependencies between SQL statements as data flow dependencies between the tables these statements operate on. Nodes in the graph represent tables, while directed edges are used to show the data flow. Edges are labeled with the operation leading to this data flow.

For each statement read from the input, our SQL2SMO tool explores its structure and identifies the affected tables, which are thus entered as the nodes of the dependency graph. Edges are due to two scenarios: a statement involves several tables, which will generate dependencies between them, or alternatively, a statement writes data into a table that another statement reads from. As an example (detailed below) if an UPDATE statement sets an attribute of table T using an attribute from table S as its source, this leads to an edge between the nodes representing S and T in the dependency graph.

Once the dependency graph has been created, its connected components (subgraphs) are computed. Each connected component represents a group of statements chained together by common input and output tables. In the following

phase, the system tries to match each statement group against the patterns in the pattern library (described in the next section). The following example, extracted from the evolution of the Ensembl database between revisions 224 and 226, illustrates the dependency graph and the connected components. For the sake of presentation, we show here only four statements manually extracted from a more complex patch. We will come back to the same example, showing how we handle the entire SQL script in Sect. 7.3 where we tackle the handling of complex SQL sequences. The sql statements are as follows:

```
(2) ALTER TABLE gene ADD description text;
(4) UPDATE gene g, gene_description gd SET
     g.description = gd.description
     WHERE gd.gene_id = g.gene_id;
(5) DROP TABLE gene_description;
(10) CREATE TABLE transcript_supporting_feature
     ( transcript_id int(11) DEFAULT   NOT NULL,
     feature_type enum(dna_align_feature,
     protein_align_feature), feature_id int(11)
     DEFAULT  NOT NULL,
     UNIQUE all_idx (transcript_id,feature_type,
     feature_id),
     KEY feature_idx (feature_type,feature_id));
```

The dependency graph extracted from this patch contains nodes *gene* (from statements (2) and (4)), *gene_description* (statements (4) and (5)), and *transcript_support_feature* (statement (10)). A single edge, *gene_description* → *gene*, is extracted from statement (4) since the *description* attribute of table *gene* is set equal to the value of the *description* attribute of table *gene_description*.

The previous dependency graph contains the two connected components:

- Component 1: *gene*, *gene_description*
- Component 2: *transcript_supporting_feature*

The second component has only statement (10) associated with it and translates immediately into a CREATE TABLE SMO.

The statements associated with the first component are statements (2), (4), and (5); these statements are matched against the pattern library to obtain a possible translation into more complex SMOs.

7.2 Patterns

A pattern is a sequence of SQL statements performing a common schema transformation that can be modeled as an SMO. The purpose of each pattern is to identify a sequence of SQL commands that is frequently used to implement a specific schema transformation.

PRISM currently recognizes the following six patterns which are thus translates into equivalent sequences of SMOs:

1. Join Pattern type 1
2. Join Pattern type 2
3. Decompose Pattern
4. Partition Pattern
5. Merge Pattern
6. Copy Pattern.

While more patterns could be easily added, our evaluation indicates that this core set of patterns captures the most used cases in our datasets. Before we enter a more detailed description of these patterns, below, let us briefly discuss how they are extracted.

The pattern-matching mechanism consists of two steps: (1) pattern-defined criteria analysis and (2) translation. The first step consists in testing several syntactic and semantic conditions (imposed by the pattern) against a sequence of SQL statements. If all conditions are met, the system generates the corresponding SMOs. Patterns are matched sequentially against the list of SQL statements, and the first matching pattern is selected for each subsequence. The design of the patterns avoids the generation of spurious SMOs.

Join Pattern Type 1

This pattern identifies a common way in which migration steps that join two tables are implemented in practice.

A Join Pattern 1 is constituted of the following SQL statements:

```
CREATE TABLE a (c1,c2...)
INSERT into a (SELECT (..) FROM b1,b2,...)
DROP TABLE b* (*optional)
```

The user performing this operation creates a new table ready to host the joined data, selects the desired columns from the source tables according to the new table's signature, and migrates the data from the old tables to the new one. Optionally, either one or both source tables can be dropped after the data migration. We have previously discussed how the pattern-matching mechanism is performed in two subphases. The first phase is further divided into additional two subphases. The first subphase is quite simple and performs a correspondence check between the statement labels and the pattern labels. The second phase is more challenging, and the long description below illustrates the details of the matching and the translation that is performed in this second phase.

To verify and confirm the correspondence between the given set of statements and the Join Pattern Type 1, the system performs the following checks:

- the INSERT statement must mention more than one table in the SELECT subclause
- the table in the CREATE and INSERT statements must be the same
- if a DROP statement is in the sequence, the table dropped must be one of the tables involved in the join.

Once the above checks succeed, the system is ready to translate the statements. The following section explains in detail some crucial operations performed by the system during the translation.

Drop table To match the SMO JOIN semantics, which consumes its input tables, the system verifies whether the tables being joined are dropped after the data have been migrated, and it introduces COPY SMOs whenever they are needed to preserve the input tables.

Columns and table rename The columns from the INSERT clause and the SELECT clause are compared looking for any possible renaming needed. If all columns are selected using the * operator, the check is performed against all the columns of the table. Each pair of columns is compared and if the names of the source and destination column differ, a RENAME SMO is generated. As the join SMO is defined creating a new table in the schema, the name given to the newly created table is the name of the table in the CREATE TABLE statement. The flexibility of the SQL syntax that allows for implicit/explicit and position or name-based references to attributes requires lots of care in our system.

Partition analysis The INSERT statement could involve all the data in the source tables or only a portion of those data. Selecting a portion of the data by using a WHERE condition in the SELECT statement corresponds, in terms of SMOs, to partition the input data. The system handles this situation looking for any condition inside the WHERE clause involving a column and a constant, where the condition can be any of <=, >=, =. Our system then generates a PARTITION SMO, dividing the partitioned tables in two subtables, one used for the join, and one discarded. The condition in the SQL WHERE clause is used to partition the data.

The following example extracted from the evolution of the Mediawiki database between the revisions 17217 and 17244 helps to understand the Join Pattern Type 1 mechanism.

```
1)CREATE TABLE redirect (rd_from int(8) unsigned
  NOT NULL, rd_namespace int NOT NULL default 0 ,
  rd_title varchar(255));
2)INSERT INTO redirect(rd_from,rd_namespace,
   rd_title)
  SELECT pl_from,pl_namespace,pl_title
  FROM pagelinks, page
  WHERE pl_from=page_id AND page_is_redirect=1;
```

The goal of this evolution step is to migrate the record containing redirected page information to the newly created table `redirect`. Once it has verified that all the pattern conditions are satisfied, the system produces the following SMOs as output:

– As both source tables are still in the schema after this evolution step, the system performs a copy of those tables, as follows:

```
COPY TABLE page INTO page_copy;
COPY TABLE pagelinks INTO pagelinks_copy;
```

– A PARTITION operation is performed next to delete the unnecessary rows from the `page` table:

```
PARTITION TABLE page_copy INTO page_copy_part_1
    WITH page_is_redirect = 1, page_copy_part_2
    WITH page_is_redirect <>1
```

– A join between the two tables is used next to obtain the desired `redirect` table:

```
JOIN TABLE page_copy_part1, pagelinks_copy
    INTO redirect WHERE pagelinks_copy.pl_from=
    page_copy_part_1.page_id
```

– Finally, the unnecessary columns are dropped from the destination table—via DROP SMOs. As the join between the page and pagelinks tables results in a table with more columns than the table `redirect`, the system drops the columns that should not be available in the final table.
– Also, for every destination column with a name different from the original one, the system will generate a RENAME SMO.

We will next introduce the remaining patterns, but for the sake of presentation, we limit the discussion to the basic SQL pattern, while the full detail of the syntactic checks and corner-case handling are available in [46].

Join Pattern Type 2

The Join Pattern type 2 covers the need of moving a table column from a source to a destination table and is denoted by the following sequential pattern of SQL statements (where '`c`' is the column being moved):

```
ALTER TABLE a ADD COLUMN c1
UPDATE TABLE a,b SET a.c1 = b.c WHERE (..)
DROP TABLE b* (*optional)
```

The final result of this operations could also have been obtained using the join of type 1, which, however, would require the creation of a new table which is not in the SQL input code examined—thus a new pattern was added. The various checks performed on this patter and its final translation into SMO are, however, similar to those of join of type 1 join, and will not be further discussed here.

The Decompose Pattern

The inverse of the join pattern is the Decompose Pattern, which is mapped into the SMO DECOMPOSE operator, manage the decomposition of a table into two subtables. The pattern is as follows:

```
CREATE TABLE a  (c1, c2, c3...)
CREATE TABLE b  (c1, c4, c5...)
INSERT INTO a SELECT (c1,c2,c3, ...) FROM c
INSERT INTO b SELECT (c1,c4,c5, ...) FROM c
DROP TABLE c* (*optional)
```

In this case, columns `c1`, `c2` and `c3` are moved from `c` to `a`, while `c1`, `c4` and `c5` are moved to `b`. Typically, `c1` is

a key, although this is not strictly required. Thus, although the decomposition operation in textbooks is handled through relational projection, in the real world, the operators actually used are insert statements. In some situations, these inserts are constrained by WHERE clauses whereby only a part of the original table `c` is decomposed. In these cases, table `c` is first partitioned horizontally, using the above-mentioned WHERE conditions. Then, the partition of the original `c` that can be reconstructed as the join of `a` and `b` is dropped without loss of information. The partition operation is discussed next.

The Partition Pattern

An SMO PARTITION operator is used to record a sequence of SQL statements that split the tuples of table between two tables, as per the following sequence:

```
CREATE TABLE a  (c1,c2...)
INSERT INTO a SELECT ... FROM c WHERE condition
DELETE FROM c WHERE condition
```

Here, the first statement creates the table `a` that will host the data moved from the source input table `c`. Then, the second statement moves the selected tuples to the new table, while the last statement deletes those tuples from `c` (thus the same WHERE `condition` is used). The merge operation described next actually provides the inverse to this operation.

The Merge Pattern

The SMO MERGE denotes a pattern of SQL statements which unions tuples of two tables into a target table. In practice, this often means merging the tuples of one table into the other one, hence the following SQL pattern:

```
INSERT INTO a SELECT * FROM b
DROP TABLE b (*optional)
```

Here, the source table `b` and destination table `a` must have the same signature. The system also accounts for scenarios in which projection of the two tables is used, and where only a subset of the columns in the target table is filled from the input tables. This implies a series of column drops and padding with null/default values. We omit this for the sake of presentation, but more details are available in [46].

The Copy Pattern

An SMO COPY operator is used to record a sequence of SQL statements that clone an existing table. While this is not very common as a standalone event, it can be used, together with rename table patterns, to account for copy-based migration, as mentioned at the beginning of this section.

The pattern of SQL statements is as follow:

```
CREATE TABLE a  (c1,c2...)
INSERT INTO a SELECT ... FROM c
```

Thus, the first statement creates the table `c` that will host the data moved from the source input table `a`. Then, the second statement moves the selected tuples to the new table. Once again projections and partitions might come into play and are handled similar to the other patterns.

Next, we discuss how multiple patterns can be tangled in an SQL sequence, and how our system handles that.

7.3 Complex SQL sequences

As previously mentioned, the SQL script that implements an evolution step might combine and mix the statements related to multiple SMOs together, thus complicating the pattern detection process. To handle this scenario, our system considers possible matches on subsets of the connected components of each SQL script.

We illustrate this with an example based on the statements publicly released on the schema history of the Ensembl database between schema revisions 224 and 226. The statements in the script are as follows (a numeric reference has been assigned to each statement):

```
(1) ALTER TABLE gene CHANGE type biotype
   VARCHAR(40) NOT NULL default   protein_coding  ;
(2) ALTER TABLE gene ADD description text;
(3) ALTER TABLE gene ADD source VARCHAR(20)
   NOT NULL default   ensembl  ;
(4) UPDATE gene g, gene_description gd
   SET g.description = gd.description
   WHERE gd.gene_id = g.gene_id;
(5) DROP TABLE gene_description;
(6) ALTER TABLE transcript
   ADD biotype VARCHAR(40)
   NOT NULL DEFAULT   protein_coding  ;
(7) ALTER TABLE transcript ADD description text;
(8) UPDATE transcript t, xref x
   SET t.description = x.description
   WHERE t.display_xref_id = x.xref_id;
(9) UPDATE transcript t, gene g
   SET t.biotype = g.biotype
   WHERE g.gene_id = t.gene_id;
(10) CREATE TABLE transcript_supporting_feature
   ( transcript_id int(11) DEFAULT   0   NOT NULL,
   feature_type enum(  dna_align_feature  ,
     protein_align_feature  ),
   feature_id int(11) DEFAULT   0   NOT NULL,
   UNIQUE all_idx (transcript_id,feature_type,
   feature_id),
   KEY feature_idx (feature_type,feature_id));
```

The first operation performed by the system is the computation of the dependencies graph as follows:

- Component 1: gene, gene_description, transcript, xref
- Component 2: transcript_supporting_feature

The second component, as in the previous example, has only statement (10) associated with it, and this statement can be easily matched and translated into a CREATE TABLE SMO. The statements that identify the first component are (1)–(9). Considering this whole group in its given order, it becomes difficult to automatically identify the operation undergone by

the schema. However, by matching patterns against subsets of this group of statements, the system identifies three fixed patterns. The first one is composed of the statements (2), (4), and (5) as follows:

```
ALTER TABLE gene ADD description text;
UPDATE gene g, gene_description gd
   SET g.description = gd.description
   WHERE gd.gene_id = g.gene_id;
DROP TABLE gene_description;
```

The subset of statements is identified and translated as a join, by matching the Join pattern of Type 2. The second pattern identified in the power set is composed of statements (7) and (8), while the last pattern is composed of the statements (6) and (9). Both sequences are identified and translated as join pattern of Type 2. From the previous analysis, we conclude that the statements not involved in any pattern are statements (1) and (3). These represent independent operations over the schema, each of which can be translated into an ADD COLUMN SMO. Notice how the group of statements has been disentangled into five standard SMOs (of which three are non-trivial JOINs).

The tool we provide is semi-automatic inasmuch the resulting translation is always presented to the user for validation. The system provides a simple interface to curate the proposed SMOs and to create new ones for the SQL statement the system could not translate.

8 Experimental evaluation

We present a series of experiments validating the PRISM++ rewriting technology. In Sect. 8.1, we test (i) the scalability and efficiency of our rewriting technique against both synthetic and Ensembl-based use cases, (ii) the overhead of our rewriting technique for a practical case based on the latency-sensitive queries from a Wikipedia workload, and (iii) the applicability of our caching techniques using traces from the Wikipedia profiler. In Sect. 8.2, we evaluate the efficiency and effectiveness of our SQL-to-SMO translation tool, using several other real-world evolution histories.

8.1 Rewriting technology

In the following, we report an evaluation of the system against actual evolution histories from [19] and synthetic cases.

The experiments have been conducted on a system with the HW/SW configuration shown in Table 4. The more powerful machine has been used to evaluate the overhead of query rewriting w.r.t. to query execution.

Among the many evolution histories we introduced in Sect. 2, we selected the two representative test cases of Wikipedia and *Ensembl* DB. The choice was due to: (i) their popularity and ii) the fact that for these two systems, we have

Table 4 Experimental environments

Machine 1	
CPUs	Quad-Core Xeon 1.6 GHz (x2)
Memory	4 GB
Hard disk	3 TB (500 GB x6), RAID-5
OS distribution	Linux Ubuntu server 6.06
OS kernel	Linux 2.6.15-54 server
Java	Sun Java 1.6.0-b105
Machine 2	
CPUs	Quad-Core Xeon 2.26 GHz (x2)
Memory	24 GB
Hard disk	6 TB (2 TB x6), HW RAID-5
OS distribution	Linux Ubuntu server 9.10
OS kernel	Linux 2.6.31-19 server
Java	Sun Java 1.6.0_20-b02

the complete database content and traces of execution for the actual production workloads—a log of 10 % of the access to the actual Wikipedia website for almost 4 months, and a complete log of the workload generated by hundreds of biologists against the *Ensembl* DB [28] for over 2 months.[15]

To test the practical relevance of our system, we tested a set of 120 SQL statements (queries and updates) from the actual workloads of Wikipedia and *Ensembl*, (i) against each operator (SMO and ICMO), (ii) through short artificial sequences of operators, and (iii) through portions of the evolution histories of Wikipedia and *Ensembl*. The system found a correct rewriting, whenever one existed, in all our tests.

Rewriting time for updates. An important measure of performance of our system is the rewriting time for updates (which subsumes that of queries). This has been the target of various optimization efforts. In Fig. 12a, we present the rewriting time of a typical set of update statements (a mix of updates, deletes, and inserts) against a portion of *Ensembl* evolution history. The test is performed on the most recent portion of the history, which contains some of the most relevant evolution steps of a public copy of the database [28] that we monitored.

The figure depicts (i) a baseline approach (which already accounts for the compression technique, and the optimized version of the *UpdateRewrite* algorithm), (ii) the effect of our Pruning technique, (iii) the averaged impact of the template-based cache, and (iv) the results of applying all of these optimizations. This combination of optimizations delivers up to 4 orders of magnitude of improvement.

Effect of chains of foreign keys The newly introduced support for integrity constraints raises a new challenge to the

[15] We release the two datasets at: http://db.csail.mit.edu/wikipedia/ and http://db.csail.mit.edu/ensembldb/.

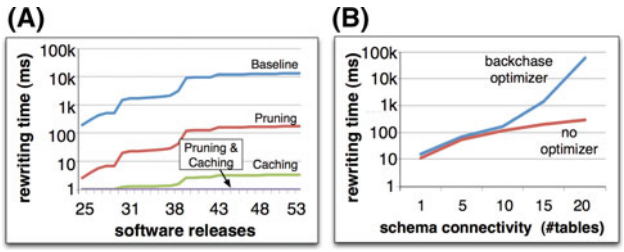

Fig. 12 **a** Rewriting scalability versus schema connectivity, **b** Averaged update rewriting time on *Ensembl* schema evolution

scalability our approach. Schemas containing large numbers of foreign keys prevent us from pruning aggressively since larger portions of the schema (those reachable via foreign keys) might be relevant for the rewriting. This leads to larger input (constraints+mappings) to the chase.

We set up a synthetic scenario in which we artificially increase the number of foreign keys, and thus the number of tables reachable from the query footprint—multiple schema layouts have been tested as discussed below. Figure 12b shows how the rewriting time grows for increasing levels of connectivity of the schema. The chase engine we use for rewriting is also used to optimize the output query (by means of a procedure known as back-chase [27]). The goal is reducing of the rewritten query/update's execution time. We show the running time of the system with and without the optimizer turned on. Both solutions are acceptable for the typical schemas from [19] (typical average connectivity <5), while the price of optimization becomes evident for highly connected schemas.

The results reported in Fig. 12b are based on the following experiment. We tested with five simple queries (results for updates are similar since they rely on the same algorithm) averaging the results for each structural SMOs (ICMO rewriting is not based on the chase and is thus not affected by the foreign keys). We first verified that the actual schema layout is not relevant to the rewriting performance, that is, having N tables directly reachable with a single-hop from the query footprint or N tables reachable through a long chain of foreign keys will lead to the same rewriting performance. We then synthetically generated several schemas with mixed properties (few long chains and few directly reachable tables) but with increasing numbers of tables reachable from the query footprint. The number of reachable tables directly influences the size of the mapping, expressed as DEDs, that we feed into the chase engine MARS. Rewriting times are given both for the scenario in which we use back-chase to improve the output query quality and when no query optimization is performed. Thanks to the nature of the backchase-based optimizer we utilize [27] it is possible to achieve partial optimization by using a subsets of the available constraints, thus achieving a trade-off between output query optimization and rewriting time.

Table 5 Overhead of rewriting

Statements	Execution time (ms)	Rewriting time (ms)	Overhead (%)
S1	77.37	1	1.29
S2	21.674	1	4.6
S3	48.2	1	2.07

End-to-end validation We assess the practical applicability of our system and the effectiveness of our caching scheme on the workload of Wikipedia (using the actual workloads from the Wikipedia on-line profiler). The system achieves an average overhead of rewriting of about 1 ms thanks to: (i) the various optimizations of the rewriting engine, (ii) a cache hit time of <1 ms, and (iii) an *extremely* high hit/miss ratios (> 5 k for updates and > 500 k for queries) due to the fact that queries/updates are automatically instantiated by the application from a small number of templates. This allows the system to amortize the cost of rewriting across many query/update executions. In order to measure the relative overhead of our solution with respect to execution time, we randomly selected 3000 instances of 3 of the most common queries from the Wikipedia workload, and tested their running time on a locally installed copy of English Wikipedia—about 3.6 TB of data.

Table 5 shows that the overhead of rewriting queries is negligible, and due to longer execution times and comparable rewriting times, the impact on updates is even less significant (typically <0.1 %). This shows that our system delivers performance that is usable even for latency-critical systems such as Wikipedia.

Below, we report the three queries used for testing execution performance. **S1:** *The query fetching the textual content of an article:*

```
SELECT old_text,old_flags
FROM text
WHERE old_id = "x"
LIMIT 1;
```

S2: *The query fetching all the metadata of a certain revision of an article):*

```
SELECT rev_id,rev_page,rev_text_id,
      rev_timestamp,rev_comment,
      rev_user_text, rev_user,
      rev_minor_edit,rev_deleted,
      rev_len, rev_parent_id,
      page_namespace,page_title,
      page_latest
FROM page,revision
WHERE (page_id=rev_page) AND
     rev_id = "x"
LIMIT 1;
```

S3: *The query fetching the current revision of a page and its metadata given the page title):*

Table 6 Caching the Wikipedia workload

Statement type	Number of templates	Avg hit/miss ratio	Max hit/miss ratio
Update	142	5,661.21	80,870
Select	1294	248,005.41	88,740,689
Select*	610	526,096.72	88,740,689

* With improved template extraction factorizing DB names

```
SELECT rev_id,rev_page,rev_text_id,
       rev_timestamp,rev_comment,
       rev_user_text, rev_user,
       rev_minor_edit,rev_deleted,
       rev_len, rev_parent_id,
       page_namespace,page_title,
       page_latest
FROM page,revision
WHERE page_namespace = "10" AND
      page_title = "x" AND
     (rev_id=page_latest) AND
     (page_id=rev_page)
LIMIT 1;
```

Caching Effectiveness In the following, we present more details about the queries we ran and the cache effectiveness.

The total number of query and update templates is typically rather small (less than a thousand for Wikipedia); therefore, the cache substitution policy (configurable and LFU by default) is not central for performance since all of the templates typically fit in main memory. The cache hit/miss ratio (shown in Table 6) and cache hit time we measured (< 1ms) for the Wikipedia dataset are very encouraging. These results are obtained for the online profiler of Wikipedia http://tinyurl.com/wikipediaprofiler.

8.2 SMO extraction performance and usability

In this section, we present the evaluation of our SMO extraction tool against the evolution histories of 13 systems, among them the one in our benchmark (Table 1). Our tool can automatically parse the MySQL dialect of SQL (and the core of standard SQL), hence the exclusion of two evolution histories leveraging other proprietary SQL dialects or unsupported constructs. Generalizing our tool to ingest arbitrary SQL dialects is beyond the scope of this paper.

We tested with a group of 4 master students using our operators to model the evolution histories of several real-world systems. Within a few hours, the students were able to learn the SMO-ICMO language and to precisely model hundreds of evolution steps from real-world systems.

To evaluate the efficiency and quality of our SMO translation, we grouped the four grad students (at this point all familiar with both SQL and SMOs), into two groups, one manually extracting SMOs directly from SQL scripts and the other leveraging our SQL to SMO extraction tool—thus

Table 7 Automatic SMO extraction

	Atutor	Coppermine	Deki	E107	Ensembl	KT-DMS	Mediawiki	Nucleus	Phpwiki	Tikiwiki	Typo3	Xoops	Zabbix	Average
Effective evol. steps	156	24	14	14	24	83	141	18	8	13	30	4	36	**43**
# of SQL statements	1168	44	124	20	174	1074	226	237	20	53	455	31	654	**329**
Fully translated	115	23	10	12	17	66	131	12	8	10	25	3	16	**34**
Partially translated	30	1	4	0	7	12	9	6	0	3	4	1	20	**7**
Not translated	11	0	0	2	0	5	1	0	0	0	1	0	0	**2**
Generated SMO	984	43	76	17	122	918	319	41	18	40	442	30	180	**248**
% Fully trans. steps	74 %	96 %	71 %	86 %	71 %	80 %	92 %	67 %	100 %	77 %	83 %	75 %	44 %	**78 %**
% Partially trans. steps	93 %	100 %	100 %	86 %	100 %	94 %	99 %	100 %	100 %	100 %	97 %	100 %	100 %	**98 %**

supervising and curating the SMOs generated by the tool. The students performed SMO extraction of almost 1,000 evolutions steps (over 4,200 SQL statements) from the evolution histories of the systems in Table 7. The resulting SMOs were equivalent, with the exception of a handful of evolution steps in which the students using the tool correctly identified certain SMOs that were missed by the control group. The more significant result was the efficiency boost enjoyed by the students using the tool: while our experimental setting was not rigorous enough to make any strong claim, the self-reported times to complete the task suggest a 3-5X reduction in the average time to complete the translation of an evolution step.

In Table 7, we also report the results of comparing the SMOs generated by the tool (before the student curation), with the manually generated ones. The tool is designed to be very conservative in generating translations, since false positives are very dangerous. In our tests, the tool had 100 % precision; hence, all SMOs automatically extracted by the tool were always correct. However, we cannot provide any formal guarantees for our heuristics, and therefore we recommend that in practical scenarios this tool be used as semi-automatic support for manual evolution.

As for recall, the tool completely automates on average 78 % of the evolution steps of our histories and (at least partially) automates an average of 98 % of the evolution steps. These results are encouraging, since the user is required to manually inspect only a small fraction of the evolution steps, for which he often has a partial translation to start with, as informally demonstrated by the efficiency boost we observed in our small user study.

As a further confirmation of the relevance of this extraction tool, we manually inspected the set of SMOs extracted for Mediawiki and Ensembl (our richest histories) and found 8.15 and 14.75 % of SMOs, respectively, were derived from non-trivial operators extracted via patterns.

It is important to note that the patterns we use have been defined manually before inspecting the evolution histories. The design of the patterns has been driven by considering the most straightforward ways to achieve the transformations captured by SMOs, and using only Ensembl as a guiding example. The quality of our results on the remaining benchmarks is comparable to the results we obtained in Ensembl. Furthermore, the tool allows users to define new patterns to increase matching performance.

To build this tool, we had to parse SQL (both DML and DDL), and we thus extended an existing Java SQL parser, to cover the richer syntax we encountered in our experiments. The by-product of this work is a parser that covers a significant subset of SQL, including (due to other uses, we made of this parser) the recent Temporal SQL extensions. The parser is released as an LGPL open-source project at: http://code.google.com/p/tsqlparser/.

9 Related work

A shorter version of this paper has appeared in [20].

Our work shares its motivation with research on inverting [31,32] and composing [30,40] schema mappings: inversion is needed to virtually migrate data back from the current schema to the old one, and composition is needed to do so over several steps in the evolution history. The main difficulty in these works stems from the expressive power of schema mappings, which leads to the non-existence of a unique migrated database. This requires evaluating queries under the certain answer semantics over all possible ways to migrate the database. This evaluation requires materializing a representative of these possible databases (known as a universal solution), and thus does not scale to the long evolution histories in our scenarios. In contrast, our approach forces a unique way to migrate the database (both forward and backward) by asking the DB administrator at evolution time to pick a migration/inversion policy. This allows standard query answering semantics, and better yet, it allows us to evaluate legacy queries and updates without migrating data back, by using rewriting instead. [31,32] do not consider updates and integrity constraints.

Other related research includes mapping adaptation [54,57] and rewriting under constraints [26,27]. However, these works do not consider update rewriting or integrity constraint editing. Different approaches have addressed the schema evolution problem from several vantage points. An incomplete list includes the methodology of [49], based on the use of views to decouple multiple logical schemas from an underlying physical schema that has monotonically non-decreasing information capacity—this is not suitable for our scenario since it is not compatible with evolution steps where integrity constraints are tightened nor with changes to the schema aiming at improving performance by reorganizing schema layout; the unified approach for propagating changes from the applications to the database schema of [35], focusing mainly on tracing and synchronizing the changes between applications and database, this methodology requires a significant commitment from DBAs and developers, which is in contrast to the evolution transparency that we seek in our work; the application-code generation techniques developed in [18] that, instead of shielding the applications from the evolution as we do, aim at propagating the changes from the DB to the application layer in a semi-automatic fashion; the framework for metadata model management [15,41] that exploits a mapping-like approach to address various metadata-related problems including schema evolution. Schema evolution support for historical databases have been studied in [44,45] with the focus on lossless archival of data history and efficient answering of historical queries against many schema versions. None of the above addresses updates under schema and integrity constraints evolution.

The difficult challenges posed by update rewriting, first elucidated in classical papers on view-update [14,25], have recently received renewed attention. In [17], new approaches were proposed, based on the notion of DB lenses. Recently, [37] proposed a new approach to support side-effect-free updating of views. The proposed solution is based on decoupling the physical and logical layer of a DBMS. This approach extends the class of updates that can be supported, but (i) requires an extension of existing RDBMS and (ii) support updates not implementable in the target DB. These two characteristics make it inappropriate to our goals. Our structural evolution operators (SMOs) can be broadly construed as views, which is why the notion of equivalent update can be adopted, but our performance gains are due to exploiting the actual semantics of SMOs; a reduction to the view-based treatment of these works would lead to having to solve an unnecessarily general case, which is notoriously hard. Moreover, none of the above works considers "views" given by editing integrity constraints, giving no guidance on how to handle ICMOs.

To handle the cases in which the original data violate target integrity constraints, classical theory on query answering prescribes to (virtually) migrate the original instance into a *set of possible worlds*, each satisfying the additional constraint and corresponding to a "repair" of the inconsistent original [12]. Repairs are usually defined to be as economical as possible, by adding/deleting the minimal number of tuples required to achieve consistency. Queries are then answered under so-called *certain answer semantics*, which is an attempt to completely automate the query answering process, by treating all minimal repairs as equally desirable and accepting only those query/update results supported by *all* repairs.

One of our contributions enabling a pragmatic system is the design decision to part with the classical set-of-repairs/certain answers semantics. For one, viewing all repairs as equally desirable is not always appropriate in practice, depending on the application. Moreover, query answering under certain answer semantics is intractable in most cases (co-NP-hardness in the size of the database is a frequently occurring lower bound in various flavors of the problem [10,12,13]). Therefore, PRISM++ allows the database administrator to pick from a list of pre-defined repair policies that are preferentially employed in practice. Each policy yields a single canonical repair. The canonical repair may not be minimal, but it is prevalent in practice and supports query answering under standard, *tractable* semantics. We show how to solve the query and update rewriting problems for the canonical repair semantics, for expressive queries and updates. This required proving new formal results, as existing work only pertains to the certain answer semantics.

Other interesting approaches to support evolution include the co-evolution framework presented in [50], where changes made to the object model are propagate via proper updates of the mapping and underlying database schema, and [47] a graph-theoretic approach to support what-if scenarios and analysis of impact of evolution.

Recently, the notion of SMOs and schema evolution in general have been studied in the context of column databases [39], and data stream management systems [51]. With the increasing attention for schema evolution support, [48] proposes a system called CRIUS that efficiently supports schema evolution and data migration through a nested data model.

Several administration tools are available today to support common management tasks, a recent survey appeared in [33]. To the best of our knowledge, none of them supports schema evolution completely, as we show in the side-by-side comparison presented in Table 8, of some of the main systems available on the market.

The table reports the capabilities of each system to document (Doc) changes to the various DB objects (schema, data, queries, updates, indexes, etc), estimate (Predict) what will be the impact of an evolution step on them, automatically adapt (Transform) various DB objects to reflect the evolution step, invert the evolution process (Reverse), for example, migrating data back or generating inverse schema transformations. Question marks indicate feature/system combinations for which we could not find enough evidence on whether they are supported or not. As shown in the table, the existing approaches support some of the basic features, but fail in providing a complete end-to-end support. In particular, all the existing tools provided by DBMS vendors or open-source efforts are focused on documenting and supporting the schema definition and the data migration, but fail short at supporting queries and updates. The documentation and data migration capabilities of PRISM++ (not discussed in this paper) are similar or superior to the one provided by some of the other tools, while the query and update rewriting technology is not available in any system we were able to test.

10 Conclusions

Modern information systems need effective methods and systems that support and automate the complex operations required by database schema upgrades. These methods and tools are bound to be successful only to the extent that they are complete and can automate the database migration and application rewriting in most real-life situations. As described in this paper, PRISM++ has tackled the hardest technical problem in achieving such completeness: that is the problem of supporting (i) integrity constraints evolution and (ii) automating query and update rewriting through structural schema changes and integrity constraints evolution. PRISM++ starts with a set of SQL-based operators called SMOs+ICMOs: although these operators are simple, they proved be sufficiently expressive to capture long evolution histories of representative real-life applications

Table 8 Schema evolution tools comparison

	DB2 CM expert [1]	Oracle CM pack [6]	MySQL work -bench [5]	IDERA SQL CM [3]	Embarcardero CM [2]	RedGate [7]	DTM DB suite [8]	SwisSQL [9]	Liquibase [4]	PRISM++
Schema										
Doc	✓	?	✓	✓	✓	✓	?	✓	✓	✓
Predict	✓	✓	×	✓	?	?	✓	×	×	✓
Transform	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Reverse	✓	✓	✓	✓	✓	✓	?	✓	✓	✓
Doc	?	?	✓	×	×	✓	?	✓	✓	✓
Data										
Predict	✓	✓	?	×	×	?	✓	×	×	✓
Transform	✓	✓	✓	×	×	✓	✓	✓	✓	✓
Reverse	?	?	✓	×	×	✓	?	?	✓	✓
Query										
Predict	×	×	×	×	×	×	×	×	×	✓
Transform	×	×	×	×	×	?	×	×	×	✓
Update										
Predict	×	×	×	×	×	×	×	×	×	✓
Transform	×	×	×	×	×	×	×	×	×	✓
Indexes, Triggers										
Predict	×	×	×	×	×	×	×	×	×	×
Store Proc., etc.										
Transform	×	×	×	×	×	×	×	×	×	×

(EnsembleDB and Wikipedia). ICMOs are key to this result, since integrity constraint editing turns out to constitute a large percentage of the evolution steps in these applications. The PRISM++ operators provide the DB administrator with a fine-grained evolution control mechanism; moreover, they provided us with the controlled settings we needed to solve the update rewriting problem. Thus, a very significant result presented in this paper is that, by using a divide-and-conquer approach in this controlled environment, we can avoid the notoriously intractable general cases of the view-update problem studied in the literature. The soundness and effectiveness of the new approach was then validated by (i) the development of a robust and efficient prototype [21] and (ii) its testing on a large schema evolution testbed [19].

The construction of the schema evolution testbed [19] was originally motivated by the need to validate the practical completeness of PRISM++ in supporting most, if not all, real-life examples of schema upgrades. While it has served this purpose, the testbed [19] is now growing into a large curated collection of schema histories which is conducive to many applications. One such application is to devise a good documentation for database schema histories. For instance, historical extensions of the current DBMS information schemas [24] can be enhanced with the SMOs describing the mappings between successive versions. The resulting meta-database can be invaluable for the DBAs, and it is also needed to explain the provenance of the data migrated from old versions to new ones. In this paper, we have discussed the tools used to derive the interesting evolution histories in our testbed. However, once PRISM is deployed in support of a given IS, it will provide this documentation automatically.

Many information systems require the archival of historical (transaction-time) data, which for archival quality must be preserved under the schema for which they were originally generated. Then, the preservation of the schema history and of the SMO+ICMO mappings between versions becomes essential to (i) map back queries expressed on current schemas to equivalent queries on the old schema and (ii) organize the results returned to users in ways that are consistent with their current schemas.

In conclusion, by supporting integrity constraints and updates, PRISM++ has removed a major obstacle toward the complete automation of DB schema upgrades. This will ensure major reductions in downtimes and efforts by DBAs and programmers. But in addition to providing these benefits, the PRISM++ project is contributing with (i) better support for schema history and data provenance, (ii) database archival and historical queries, and (iii) a comprehensive testbed of significant schema histories. While work on (ii) has already reached its main research goals [44], work is still in progress on objectives (i) and (iii), and will be documented in future reports.

Acknowledgments The authors would like to thank Fabrizio Moroni, MyungWon Ham for their help in developing the tool, and Letizia Tanca for the great feedback and support.

References

1. http://publib.boulder.ibm.com/infocenter/mptoolic/v1r0/index.jsp?topic=/com.ibm.db2tools.chx.doc.ug/chxucoview01.htm
2. http://www.embarcadero.com/products/db-change-manager
3. http://www.idera.com/SQL-Server/
4. http://www.liquibase.org/
5. http://www.mysql.com/products/workbench/
6. http://www.oracle.com/us/products/enterprise-manager/change-management-pack-11g-ds-068451.pdf
7. http://www.red-gate.com/
8. http://www.sqledit.com/index.html
9. http://www.swissql.com/
10. Abiteboul, S., Duschka, O.M.: Complexity of answering queries using materialized views. In: PODS, pp. 254–263 (1998)
11. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison Wesley, Reading (1995)
12. Afrati, F.N., Kolaitis, P.G.: Repair checking in inconsistent databases: Algorithms and complexity. In: ICDT, pp. 31–41 (2009)
13. Arenas, M., Bertossi, L., Chomicki, J.: Consistent query answers in inconsistent databases. In: PODS, pp. 68–79 (1999)
14. Bancilhon, F., Spyratos, N.: Update semantics of relational views. ACM Trans. Database Syst. **6**(4), 557–575 (1981)
15. Bernstein, P.A.: Applying model management to classical meta data problems. In: CIDR (2003)
16. Bernstein, P.A., Green, T.J., Melnik, S., Nash, A.: Implementing mapping composition. VLDB J. **17**(2), 333–353 (2008)
17. Bohannon, A., Pierce, B.C., Vaughan, J.A.: Relational lenses: A language for updatable views. In: PODS, pp. 338–347 (2006)
18. Cleve A., Hainaut, J.-L.: Co-transformations in database applications evolution. In: GTTSE, pp. 409–421 (2006)
19. Curino, C., Ham, M., Moroni, F., Zaniolo, C.: Pantha rei data set. http://data.schemaevolution.org/ (2009)
20. Curino, C., Moon, H.J., Deutsch, A., Zaniolo, C.: Update rewriting and integrity constraint maintenance in a schema evolution support system: Prism++. PVLDB **4**(2), 117–128 (2010)
21. Curino, C., Moon, H.J., Ham, M., Zaniolo, C.: The prism workbench: Database schema evolution without tears. In: ICDE (2009)
22. Curino, C., Moon, H.J., Tanca, L., Zaniolo, C.: Schema evolution in Wikipedia: Toward a web information system benchmark. ICEIS (2008)
23. Curino, C., Moon, H.J., Zaniolo, C.: Graceful database schema evolution: The prism workbench. PVLDB **1**(1), 761–772 (2008)
24. Curino, C., Moon, H.J.: C. Zaniolo. Managing the history of metadata in support for db archiving and schema evolution. In: ECDM (2008)
25. Dayal, U., Bernstein, P.A.: On the correct translation of update operations on relational views. ACM Trans. Database Syst. **7**(3), 381–416 (1982)
26. Deutsch, A., Nash, A., Remmel, J.: The chase revisited. In: PODS, pp. 149–158 (2008)
27. Deutsch, A., Tannen, V.: Mars: A system for publishing xml from mixed and redundant storage. In: VLDB, pp. 201–212 (2003)
28. Ensembl development team. Ensembl Genetic DB http://www.ensembl.org, 2009 (Online)
29. Fagin, R.: Inverting schema mappings. ACM Trans. Database Syst. **32**(4), 25:1–25:51 (2007)
30. Fagin, R., Kolaitis, P.G., Popa, L., Tan, W.-C.: Composing schema mappings: Second-order dependencies to the rescue. ACM Trans. Database Syst. **30**(4), 994–1055 (2005)
31. Fagin, R., Kolaitis, P.G., Popa, L., Tan, W.-C.: Quasi-inverses of schema mappings. In: PODS, pp. 123–132 (2007)
32. Fagin, R., Kolaitis, P.G., Popa, L., Tan, W. C.: Reverse data exchange: Coping with nulls. In: PODS, pp. 23–32 (2009)
33. Hartung, M., Terwilliger, J.F., Rahm, E.: Recent advances in schema and ontology evolution. In: Schema Matching and Mapping, pp. 149–190 (2011)
34. Hernández, M.A., Miller, R.J., Haas, L.M.: Clio: A semi-automatic tool for schema mapping. In: SIGMOD, p. 607 (2001)
35. Hick, J.-M., Hainaut, J.-L.: Database application evolution: A transformational approach. Data Knowl. Eng. **59**(3), 534–558 (2006)
36. Hull, R.: Non-finite specifiability of projections of functional dependency families. Theor. Comput. Sci. **39**, 239–265 (1985)
37. Kotidis, Y., Srivastava, D., Velegrakis, Y.: Updates through views: A new hope. In: ICDE, p. 2 (2006)
38. Lenzerini, M.: Data integration: A theoretical perspective. In: PODS, pp. 233–246 (2002)
39. Liu, Z., He, B., Hsiao, H.-I., Chen, Y.: Efficient and scalable data evolution with column oriented databases. In: EDBT (2011)
40. Madhavan, J., Halevy, A.Y.: Composing mappings among data sources. In: VLDB, pp. 572–583 (2003)
41. Melnik, S., Rahm, E., Bernstein, P.A.: Rondo: A programming platform for generic model management. In: SIGMOD (2003)
42. Miller, R.J., Ioannidis, Y.E., Ramakrishnan, R.: The use of information capacity in schema integration and translation. In: VLDB, pp. 120–133 (1993)
43. Miller, R.J., Ioannidis, Y.E., Ramakrishnan, R.: Schema equivalence in heterogeneous systems: Bridging theory and practice. Inf. Syst. **19**(1), 3–31 (1994)
44. Moon, H.J., Curino, C., Deutsch, A., Hou, C.-Y., Zaniolo, C.: Managing and querying transaction-time databases under schema evolution. PVLDB **1**(1), 882–895 (2008)
45. Moon, H.J., Curino, C., Zaniolo, C.: Scalable architecture and query optimization for transaction-time dbs with evolving schemas. In: SIGMOD Conference, pp. 207–218 (2010)
46. Moroni, F.: Schema Evolution Toolsuite: Analysis and Interpretation of Relational Schema Changes. Master's thesis, Politecnico di Milano—Dipartimento di Elettronica e Informazione (2009)
47. Papastefanatos, G., Vassiliadis, P., Simitsis, A., Vassiliou, Y.: Hecataeus: Regulating schema evolution. In: ICDE, pp. 1181–1184, March (2010)
48. Qian, L., LeFevre, K., Jagadish, H.V.: Crius: User-friendly database design. PVLDB **4**(2), 81–92 (2010)
49. Ra, Y.-G.: Relational schema evolution for program independency. In: Proceedings of the 7th international conference on Intelligent Information Technology, pp 273–281, Springer, Heidelberg (2004). doi:10.1007/978-3-540-30561-3_29
50. Terwilliger, J.F., Bernstein, P.A., Unnithan, A.: Worry-free database upgrades: Automated model-driven evolution of schemas and complex mappings. In: SIGMOD Conference (2010)
51. Terwilliger, J.F., Fernández-Moctezuma, R., Delcambre, L.M.L., Maier, D.: Support for schema evolution in data stream management systems. J. UCS **16**(20), 3073–3101 (2010)
52. Ullman, J.: Principles of Database System. Computer Science Press, Rockville (1982)
53. Ullman, J.D.: Information integration using logical views. Theor. Comput. Sci. **239**(2), 189–210 (2000)
54. Velegrakis, Y., Miller, R.J., Popa, L.: Mapping adaptation under evolving schemas. In: VLDB, pp. 584–595 (2003)
55. Wikimedia Foundation. Wikipedia, the free encyclopedia http://en.wikipedia.org/, 2007 (Online)
56. Wikimedia Foundation. The mediawiki http://www.mediawiki.org, 2008
57. Yu, C., Popa, L.: Semantic adaptation of schema mappings when schemas evolve. In: VLDB, pp. 1006–1017 (2005)