Update Rewriting and Integrity Constraint Maintenance in a Schema Evolution Support System: PRISM++

Carlo A. Curino
MIT
curino@mit.edu

Hyun Jin Moon
NEC Labs America
hjmoon@sv.nec-labs.com

Alin Deutsch
UCSD
deutsch@cs.ucsd.edu

Carlo Zaniolo
UCLA
zaniolo@cs.ucla.edu

ABSTRACT

Supporting legacy applications when the database schema evolves represents a long-standing challenge of practical and theoretical importance. Recent work has produced algorithms and systems that automate the process of data migration and query adaptation; however, the problems of evolving integrity constraints and supporting legacy updates under schema and integrity constraints evolution are significantly more difficult and have thus far remained unsolved. In this paper, we address this issue by introducing a formal evolution model for the database schema structure and its integrity constraints, and use it to derive update mapping techniques akin to the rewriting techniques used for queries. Thus, we (i) propose a new set of Integrity Constraints Modification Operators (ICMOs), (ii) characterize the impact on integrity constraints of structural schema changes, (iii) devise representations that enable the rewriting of updates, and (iv) develop a unified approach for query and update rewriting under constraints. We then describe the implementation of these techniques provided by our PRISM++ system. The effectiveness of PRISM++ and its enabling technology has been verified on a testbed containing evolution histories of several scientific databases and web information systems, including the Genetic DB Ensembl (410+ schema versions in 9 years), and Wikipedia (240+ schema versions in 6 years).

1. INTRODUCTION

Practitioners and researchers have long acknowledged the problem of evolving Information Systems to respond to perpetually changing requirements [24]. As a result of past database research, practitioners can now rely on sophisticated methods and robust tools for designing their initial schema; however, effective methods and tools for supporting and automating the later unavoidable evolution of the schema are still missing from practitioner's arsenal. The need for practical solutions already very strong in traditional enterprise environments is made even more urgent by the growing popularity of large web information systems such as Wikipedia (over 240 schema versions in 6 years) and data-intensive, Big-Science projects such as $Ensembl$ Genome project [15] (over 410 schema versions in 9 years) or the LHC project at European Organization for Nuclear Research (CERN)[1]. The large number and diversity of stakehold-

---

[1]See: http://cern.ch.

ers, and the highly collaborative, fast-progressing environment that is typical of today's enterprise and web and scientific endeavors, is characterized, in fact, by very strong need for evolution, and reduced tolerance for migration downtime. These conclusions are also supported by the histories of 12 major information systems collected in our schema evolution testbed [10]. This analysis also confirms the need to support integrity constraints and updates: as examples consider the $Ensembl$ DB schema history containing over 175 individual changes of primary and foreign keys and that almost 20% of the SQL statements in the Wikipedia workload are legacy updates.

Until today, this manifest need for schema evolution support remained largely unanswered, even though much progress was made on topics, such as mapping composition, invertibility and query rewriting [17, 16, 13] that provide the formal basis for sound solutions. The computational costs of these techniques in their general form has 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 [7, 11, 30].

In fact, the common practice today 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.

To the best of our knowledge, PRISM++ is *the first system that offers end-to-end support for query and update adaptation through both structural schema changes and integrity constraints evolution.* The design of PRISM++ benefits from (i) our previous experience [11] and our schema evolution testbed [10], and (ii) a careful analysis and avoidance of computational pitfalls that pervade this domain.

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[2]:

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 $U(I)$, then migrates $U(I)$ "forward" through the evolution steps, to obtain a new S'-instance, replacing I'.

---

[2]This is an adaptation of the classical view update semantics [5, 12, 22] to our context, in which evolution operators replace the views.
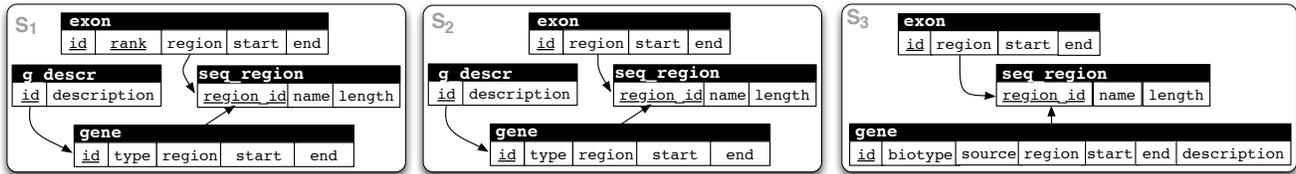
Figure 1: Three (simplified) schema versions from the actual $Ensembl$ **genetic DB schema history.**

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'.

Our first attempt in this direction [11], lacked the support of updates, was not designed to handle evolution steps modifying integrity constraints, and could only rewrite a limited class of queries.

PRISM++ solves all this by introducing *update rewriting* to adapt legacy updates to run on the current schema, *evolution of integrity constraints* significantly extending the class of evolution steps covered, and finally provides support for 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 [5], deciding schema equivalence [28], schema mapping composition [16] and inversion [17], and consistent query answering [3]. The most general version of the schema evolution problem modeled under these formalisms tend to be intractable or even undecidable (for schema mappings expressed classically, in the language of arbitrary views [31] or of source-target tgds [19, 23])—see Section 6 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 [10] provided us with the ability of testing 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 on-line[3].

1.1 Running Example: a Genetic DB

The PRISM++ system has been designed and validated on many evolution histories from several application domains [10], among which we chose the genetic DB $Ensembl$ as running example.

The $Ensembl$ project[1], funded by the European Biology Institute and the Welcome Trust Sanger Institute, provides a data-centric plat-

form 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 to 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 maintain. Hence, there is a substantial need for transparent evolution support.

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 Figure 1 is an excerpt of the CVS[4] schema revision 188.2.6; this schema describes how the $Ensembl$ DB stores its information about DNA sequences, exons[5] 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[6], leading to the schema S_2 in Figure 1 (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 Figure 1 (revision 226 of CVS schema). This example is used throughout the paper to illustrate our technical contributions.

The remainder of this paper is organized as follows: Section 2 presents the evolution language, Section 3 describes the resulting data migration, Section 4 details query and update rewriting, Section 5 discusses optimizations and experiments. Section 6 and 7 summarize related works and conclusions.

2. A SCHEMA EVOLUTION LANGUAGE

In [11] we introduced the Schema Modification Operators (SMO) of Table 1. Each operator captures an atomic (and natural) change performed to evolve the schema. By combining them, it is possible to express complex evolutions. The SMOs' atomicity and clear semantics represent an ideal basis to tackle the problem of data migration and schema evolution. However, SMOs alone do not capture integrity constrains evolution. PRISM++ extends this approach by introducing six new operators used to edit the schema integrity constraints: the Integrity Constraints Modification Operators (ICMOs)

---

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

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

[5]An $exon$ is a nucleic acid sequence related to a portion of DNA.

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

shown in the second part of Table 1. 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 Section 3. PRISM++ supports three basic integrity constraints: *primary keys*, *foreign keys*, and simple *value constraints*[7]. This set of simple constraints covers all the constraints that were actually used in the large dataset of [10]. In the following, we provide 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 Section 1.1. The DBA describes the structural and integrity constraints changes as in the following:

EXAMPLE 2.1. *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 Section 3.

2.1 Impact of SMO on Integrity Constraints

Integrity constraint evolution occurs directly (when the administrator add or remove constraints via ICMOs), or indirectly (when an SMO changes a schema structure mentioned by a constraint). 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 write $IC_1 \models_M IC_2$ —see Appendix A, for details.

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. [21].

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 SMOa and ICMOs—that have been carefully designed to enable all common evolution scenarios, while avoiding complexity/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—see Appendix A.

2.2 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.

DEFINITION 2.1. *We say that an evolution operator O from schema S_1 to schema S_2 is* information preserving *if (i) O is functional, i.e. 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 [17], schema equivalence [28], information capacity [27], 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, i.e. have the same *information capacity*.

---

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

Table 1: 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*|*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 $vc1$

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 [11]. To this end, we successfully exploited ICMOs, which are by definition not information-preserving and require special handling anyway (as discussed in Sections 4.1 and 4.3).

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, finergrained tool for describing evolution—the intuitive advantage is to *separate management of structural modifications from alterations of the information capacity (i.e., IC editing)*.

This is illustrated by Example 2.2, which displays the operator sequence used to evolve schema S_2 into S_3.

EXAMPLE 2.2. *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 modifications (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. 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.

3. DATA MIGRATION

The new evolution language we designed guarantees that data migration steps through SMOs will always be invertible (and informa-
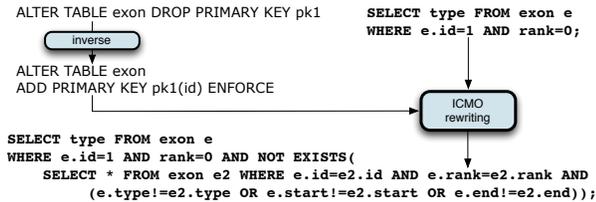
```
ALTER TABLE exon DROP PRIMARY KEY pk1        SELECT type FROM exon e
                                             WHERE e.id=1 AND rank=0;
         [ inverse ]

ALTER TABLE exon
ADD PRIMARY KEY pk1(id) ENFORCE                      ICMO
                                                   rewriting

SELECT type FROM exon e
WHERE e.id=1 AND rank=0 AND NOT EXISTS(
     SELECT * FROM exon e2 WHERE e.id=e2.id AND e.rank=e2.rank AND
          (e.type!=e2.type OR e.start!=e2.start OR e.end!=e2.end));
```

Figure 2: Query Rewriting through ICMO: ENFORCE.

tion 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 [10]. 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 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 [3]. 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 [3] 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).

4. A NEW REWRITING TECHNOLOGY

In the following, we discuss our rewriting technology. Just as a reminder, PRISM++ implements via rewriting the semantics we present in the introduction, which corresponds 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 to handle integrity constraints, updates and query with negation and functions proved to be a major technical challenge. We devote this section to the problems of rewriting queries through ICMOs and rewriting updates through ICMOs and SMOs. For lack of space, we relegate the extension to handle negation and user defined functions of our prior work on chase-based query rewriting [11] to Appendix B. 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.

4.1 Query Rewriting through ICMOs

We focus here on ICMO-based evolution steps, where 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[8]. Therefore, tightening integrity constraints (difficult for data migration) becomes trivial for rewriting.

On the contrary, relaxing the integrity constraints, e.g., 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 [10]—they correspond to special cases of the general view-update theory that have great practical appeal:

IGNORE[9]: 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[5]. 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.

---

[8]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.

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

Table 2: 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 * FROM exon
UPDATE exon SET end="342" WHERE id =1	SELECT id,type,region,start,"342" FROM exon WHERE id =1 UNION SELECT id,type,region,start,end FROM exon WHERE id !=1	= SELECT * FROM exon

CHECK: the rewriting engine checks that the DB instance I_2, satisfies the removed constraint k, e.g., in the first step of Example 2.1 if we apply 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 is returned otherwise—these conditions are implemented as probe queries, as shown later in Section 4.3 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 [10]). 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 Section 3). Let us demonstrate this, concentrating on the first operator of Example 2.1 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 Figure 2. The DBA is offered to select the enforcement policy for the inverse ICMO, ENFORCE in Figure 2. 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, i.e., 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.

4.2 Update Rewriting through SMOs

We introduce update rewriting through SMOs by means of the example in Figure 3, which demonstrates update rewriting through an evolution step decomposing table exon[10]. Figure 3 shows how the

---

[10]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)
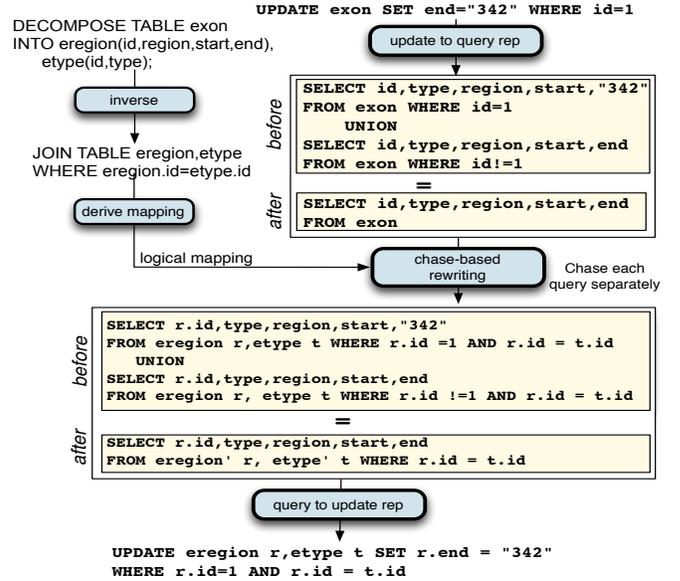

Figure 3: Update Rewriting through SMO.

PRISM++ system, in order to rewrite SQL updates: (i) represents the input SQL update in an internal format based on queries, a "trick" 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 Figure 3 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 2.

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: The rewriting algorithm: $UpdateRewrite$

Input: U_1, M'
Output: U_2
foreach *equivalence* $R \in U_1$ **do**
    $R_l = \text{left}(R)$;
    $R_r = \text{right}(R)$;
    $R'_l = \text{QueryRewrite}(R_l, M')$;
    $R'_r = \text{QueryRewrite}(R_r, M')$;
    if $R'_l = \emptyset$ *or* $R'_r = \emptyset$ **then**
        fail();
    end
    $U_2 = U_2 \cup (R'_l = R'_r)$
end

$UpdateRewrite$ rewrites each query in the equivalence independently, by means of $QueryRewrite$ (the extension handling negation of our query rewriting algorithm, summarized in Appendix B), 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 statement(s) valid on the new schema show in Figure 3—details in Appendix D.3. The resulting update satisfies the semantics from view-update literature [5, 12]:

DEFINITION 4.1. *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 4.1. *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: $U_2 = M \circ U_1 \circ M'$.*

See Appendix C for the proof of Theorem 4.1.

4.3 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, i.e., I_2^{viol} might be not empty, and might be affected by the update. This implies potential side effects, 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.

CHECK: PRISM++ checks that the constraint k is satisfied by the DB instance also *after* the update execution, i.e., $U_2(I_2) \models k$. 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 Figure 4, where we rewrite an INSERT statement through the same evolution of Figure 2, but with 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:
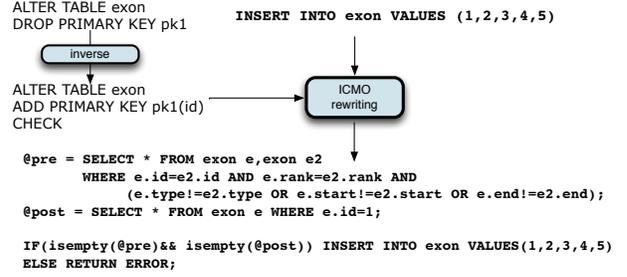$I_2^{viol} = U_2(I_2)^{viol}$.


```
ALTER TABLE exon
DROP PRIMARY KEY pk1        INSERT INTO exon VALUES (1,2,3,4,5)

      inverse                        ICMO
                                   rewriting
ALTER TABLE exon
ADD PRIMARY KEY pk1(id)
CHECK

@pre = SELECT * FROM exon e,exon e2
       WHERE e.id=e2.id AND e.rank=e2.rank AND
             (e.type!=e2.type OR e.start!=e2.start OR e.end!=e2.end);
@post = SELECT * FROM exon e WHERE e.id=1;

IF(isempty(@pre)&& isempty(@post)) INSERT INTO exon VALUES(1,2,3,4,5)
ELSE RETURN ERROR;
```

Figure 4: Update Rewriting through ICMO: CHECK.

5. OPTIMIZATION AND EVALUATION

The PRISM++ system has been implemented in Java and is loosely based on our prior system [11], 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 speed-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 procedure called the chase [13], that even in the very optimized implementation we use [14] is intrinsically expensive. 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. Thus, to achieve performance we try to contain the size of the chase input.

The key optimizations that make PRISM++ practical include: i) an adaptation of the mapping compression approach of [11] (exploiting composition to reduce the size of the chase input), ii) a mapping pruning technique, extending the basic principle sketched in [29], that removes from the input to the chase mappings and integrity constraints not relevant for the rewriting of a given query/update, iii) an optimization of the basic $UpdateRewrite$ algorithm we presented, that caches partial rewritings of the various queries it processes, and iv) a more sophisticated caching technique that caches rewritings for user queries/updates whenever they share a template (i.e., when they have similar structures but different parameters). These optimizations are discussed in details in Appendix D, while their impact on performance is discussed next.

In the following, we report an evaluation of the system against actual evolution histories from [10] and synthetic cases—Appendix E provide more details on all of these experiments. Among the many evolution histories we selected the two representative test cases of Wikipedia and $Ensembl$ DB. The choice was due to: i) their popularity and ii) to the fact that for these two systems we have the complete databases and real 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 [15] for over 2 months[11].

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 ex-

---

[11]We release the two datasets at: http://db.csail.mit.edu/wikipedia/ and http://db.csail.mit.edu/ensembldb/
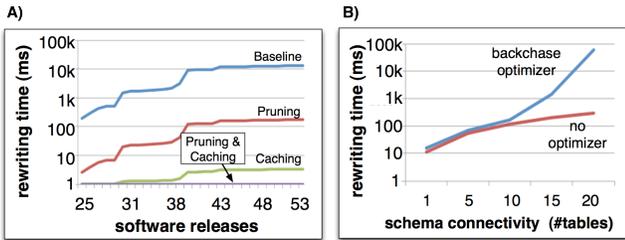
Figure 5: A) Rewriting scalability vs schema connectivity, B) Averaged update rewriting time on $Ensembl$ schema evolution

isted, 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 Figure 5A, 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, and that corresponds to some of the a public copy of the database [15] 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 deliver up to 4 orders of magnitude of improvement.

Effect of chains of foreign keys. The newly introduced support for integrity constraints introduces a new challenge to the scalability our approach. Schemas containing large number of foreign keys prevent us from pruning aggressively since larger portion of the schema (the one 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 in Appendix E.

Figure 5B 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 [14]). The goal is reducing of the rewritten query/update 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 [10] (typical average connectivity <5), while the price of optimization becomes evident for highly connected schemas.

End-to-end validation. We assess the practical applicability of our system and the effectiveness of our caching scheme on the workload of Wikipedia. The experiment is based on the actual workloads from the Wikipedia on-line profiler—details in Appendix E. The system achieves an average overhead of rewriting of about 1ms thanks to: i) the various optimizations of the rewriting engine, ii) a cache hit time of $<$ 1ms, and iii) an *extremely* high hit/miss ratios ($>$ 5k for updates and $>$ 500k 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/updates 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 test their running time on a locally installed copy of english Wikipedia— about 3.6TB of data.

Table 3 shows that the overhead of rewriting queries is negligi-

Table 3: Overhead of rewriting

Statements	execution time	rewriting time	overhead
S1	77.37 ms	1 ms	1.29%
S2	21.674 ms	1 ms	4.6%
S3	48.2 ms	1 ms	2.07%

ble, and thanks 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 are usable even for latency-critical systems such as Wikipedia.

Finally, as an informal evaluation of usability, we tested with a group of 3 master students using our operators to model the evolution histories of several real-world systems. Within a few hours the student were able to learn the SMO-ICMO language and to precisely model tens of evolution steps from real-world systems.

6. RELATED WORK

Our work shares its motivation with research on inverting [17, 18] and composing [25, 16] 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. [17, 18] do not consider updates and integrity constraints.

Other related research includes mapping adaptation [32, 33] and rewriting under constraints [13, 14]. 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 [30], 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 improve performance by reorganizing schema layout; the unified approach for propagating changes from the applications to the database schema of [20], 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 [9], 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 [26, 6], that exploits a mapping-like approach to address various metadata-related problems including schema evolution. 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 [5, 12], have recently received renewed attention. In [8], new approaches were proposed, based on the notion of DB lenses. Recently, [22] 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 [3]. 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 [1, 3, 4]). 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.

Several administration tools are available today to support common management tasks. None of them supports schema evolution completely, as we show in a side-by-side comparison in Appendix F.

7. CONCLUSIONS

PRISM++ advances the state of the art in schema evolution by supporting (i) integrity constraints evolution, and (ii) automatic query and update rewriting through structural schema changes and integrity constraints evolution.

The language of SMOs+ICMOs proves to be sufficiently expressive to capture long evolution histories of representative real-life applications (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. Our evolution language provides the DB administrator with a fine-grained evolution control mechanism, and allows us to solve the update rewriting problem in a controlled setting, using a divide-and-conquer approach, thus avoiding the notoriously intractable general cases of view update studied in the literature.

A robust and efficient prototype has been built and its soundness and performance tested on a large schema evolution testbed [10].

8. REFERENCES
[1] S. Abiteboul and O. M. Duschka. Complexity of answering queries using materialized views. In *PODS*, pages 254–263, 1998.
[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of databases*. Addison Wesley, 1995.
[3] F. N. Afrati and P. G. Kolaitis. Repair checking in inconsistent databases: algorithms and complexity. In *ICDT*, pages 31–41, 2009.
[4] M. Arenas, L. Bertossi, and J. Chomicki. Consistent query answers in inconsistent databases. In *PODS*, pages 68–79, 1999.
[5] F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Trans. Database Syst.*, 6(4):557–575, 1981.
[6] P. A. Bernstein. Applying model management to classical meta data problems. In *CIDR*, 2003.
[7] P. A. Bernstein, T. J. Green, S. Melnik, and A. Nash. Implementing mapping composition. *VLDB J.*, 17(2):333–353, 2008.
[8] A. Bohannon, B. C. Pierce, and J. A. Vaughan. Relational lenses: a language for updatable views. In *PODS*, pages 338–347, 2006.
[9] A. Cleve and J.-L. Hainaut. Co-transformations in database applications evolution. In *GTTSE*, pages 409–421, 2006.
[10] C. Curino, M. Ham, F. Moroni, and C. Zaniolo. Pantha rei data set : http://data.schemaevolution.org/. 2009.
[11] C. Curino, H. J. Moon, and C. Zaniolo. Graceful database schema evolution: the prism workbench. *PVLDB*, 1(1):761–772, 2008.
[12] U. Dayal and P. A. Bernstein. On the correct translation of update operations on relational views. *ACM Trans. Database Syst.*, 7(3):381–416, 1982.
[13] A. Deutsch, A. Nash, and J. Remmel. The chase revisited. In *PODS*, pages 149–158, 2008.
[14] A. Deutsch and V. Tannen. Mars: A system for publishing xml from mixed and redundant storage. In *VLDB*, pages 201–212, 2003.
[15] Ensembl development team. Ensembl Genetic DB http://www.ensembl.org, 2009. [Online].
[16] R. Fagin, P. G. Kolaitis, L. Popa, and W.-C. Tan. Composing schema mappings: Second-order dependencies to the rescue. *ACM Trans. Database Syst.*, 30(4):994–1055, 2005.
[17] R. Fagin, P. G. Kolaitis, L. Popa, and W.-C. Tan. Quasi-inverses of schema mappings. In *PODS*, pages 123–132, 2007.
[18] R. Fagin, P. G. Kolaitis, L. Popa, and W. C. Tan. Reverse data exchange: coping with nulls. In *PODS*, pages 23–32, 2009.
[19] M. A. Hernández, R. J. Miller, and L. M. Haas. Clio: A semi-automatic tool for schema mapping. In *SIGMOD*, page 607, 2001.
[20] J.-M. Hick and J.-L. Hainaut. Database application evolution: a transformational approach. *Data Knowl. Eng.*, 59(3):534–558, 2006.
[21] R. Hull. Non-finite specifiability of projections of functional dependency families. *Theor. Comput. Sci.*, 39:239–265, 1985.
[22] Y. Kotidis, D. Srivastava, and Y. Velegrakis. Updates through views: A new hope. In *ICDE*, page 2, 2006.
[23] M. Lenzerini. Data integration: A theoretical perspective. In *PODS*, pages 233–246, 2002.
[24] Y. Liu, S. ren Zhang, and M. qi Fang. Ecological analysis on evolution of information systems. In *I3E (2)*, pages 308–315, 2007.
[25] J. Madhavan and A. Y. Halevy. Composing mappings among data sources. In *VLDB*, pages 572–583, 2003.
[26] S. Melnik, E. Rahm, and P. A. Bernstein. Rondo: A programming platform for generic model management. In *SIGMOD*, pages 193–204, 2003.
[27] R. J. Miller, Y. E. Ioannidis, and R. Ramakrishnan. The use of information capacity in schema integration and translation. In *VLDB*, pages 120–133, 1993.
[28] R. J. Miller, Y. E. Ioannidis, and R. Ramakrishnan. Schema equivalence in heterogeneous systems: bridging theory and practice. *Inf. Syst.*, 19(1):3–31, 1994.
[29] H. J. Moon, C. Curino, A. Deutsch, C.-Y. Hou, and C. Zaniolo. Managing and querying transaction-time databases under schema evolution. *PVLDB*, 1(1):882–895, 2008.
[30] Y.-G. Ra. Relational schema evolution for program independency. *Intelligent Information Technology*, pages 273–281, 2005.
[31] J. D. Ullman. Information integration using logical views. *Theor. Comput. Sci.*, 239(2):189–210, 2000.
[32] Y. Velegrakis, R. J. Miller, and L. Popa. Mapping adaptation under evolving schemas. In *VLDB*, pages 584–595, 2003.
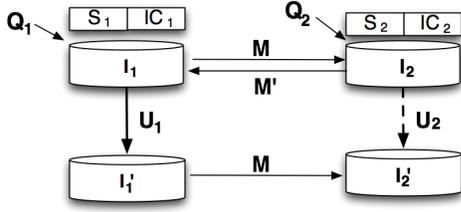[33] C. Yu and L. Popa. Semantic adaptation of schema mappings when schemas evolve. In *VLDB*, pages 1006–1017, 2005.

Figure 6: The general framework

APPENDIX

A. FORMALIZING IC IMPLICATION

With reference to Figure 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. The notion of *constraints implication* can be introduced as follows:

DEFINITION A.1. *Let IC_1 be a set of integrity constraints over schema S_1, and M a mapping from S_1 to S_2, then we write:*
$$IC_1 \models_M ic \ \ iff \ \ \forall I_1, I_2 (I_2 = M(I_1) \wedge I_1 \models IC_1 \implies I_2 \models ic)$$

The above definition 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. [12] The notion of closure is naturally obtained as:

DEFINITION A.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 1, 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 2.2 to IC_1 under the logical mapping M corresponding to the above SMO, we can determine the set of output

---

[12]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 [19] may associate several possible S_2-instances with a given S_1-instance.
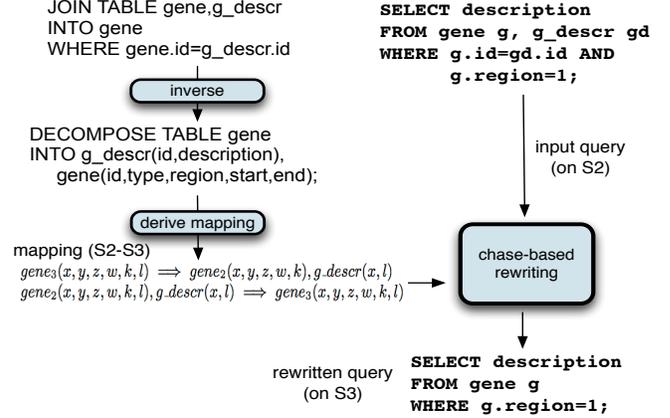

Figure 7: Query Rewriting through SMO.

integrity constraint IC_2 to be the following:

$$IC_2 : \ \ V1(a,b), \ \ V1(a,b') \implies b=b'$$
$$V2(a,c), \ \ V2(a,c') \implies c=c'$$
$$V1(a,b) \implies \exists c V2(b,c)$$
$$V2(b,c) \implies \exists a V1(a,b)$$

B. QUERY REWRITING THROUGH SMOS

In order to rewrite queries and updates through SMO-based evolution steps, the PRISM++ system: (i) inverts SMO sequences[13], (ii) translates each SMO into an equivalent logical schema mapping expressed in the language of Disjunctive Embedded Dependencies (DED) [14], and (iii) rewrites queries using these DEDs by means of a chase-based algorithm named *chase&backchase (C&B)* [14].

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 [11], and we only illustrate it by means of the example in Figure 7.

Figure 7 shows an example of rewriting through operator 3 of Example 2.2 (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 [13] to rewrite the input query into an equivalent one operating on the new schema, according to the following semantics:

DEFINITION B.1. *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 Figure 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 [11].

We can show the following (which extends the results in [11] to incorporate integrity constraints on the schemas:

---

[13]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 [11].

THEOREM B.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 [14] is guaranteed to find one.*

The acyclicity of a set of foreign keys is a classical concept [2], and a special case of the notion of weak acyclicity of a set of embedded dependencies [19]. 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 [2], which the C&B algorithm [14] relies on.

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

B.1 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 [14] 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 Figure 2 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 [13] 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.

We can prove that $QueryRewrite$ is a *sound* rewriting algorithm even when queries contain negation, or user-defined functions.

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 [10] we tested, delivering a significant improvement with respect to the state of the art.
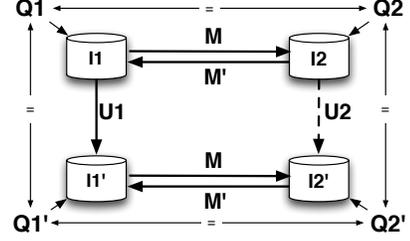
C. PROOFS

Figure 8: Update rewriting

Proof of Theorem 4.1 We refer to Figure 8. We start from an update U_1, defined in terms of queries Q_1 and Q_1' as follows:

$$U_1(I) = I' \iff Q_1(I) = Q_1'(I').$$

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, i.e.

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

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 B.1 we know that the C&B yields equivalent rewritings, i.e. $Q_2(I_2) = Q_1(M'(I_2))$, 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 B.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

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

$\square$

D. IMPLEMENTATION AND OPTIMIZATION

D.1 Speeding up rewriting

The translation from update statements to set of queries and back, and the preprocessing steps of the $QueryRewrite$ algorithm have limited impact on the overall rewriting time. However, in order to complete the rewriting $QueryRewrite$ invokes the C&B procedure for positive and negative portions of each queries produced in the translation. To reduce the cost of rewriting we re-implemented the mapping-compression technique presented in [11]. 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

Table 4: Experimental Environments

Environment	Description
CPUs	Quad-Core Xeon 1.6GHz (x2)
Memory	4GB
Hard Disk	3TB (500GB 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
CPUs	Quad-Core Xeon 2.26GHz (x2)
Memory	24GB
Hard Disk	6TB (2TB 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

Table 5: Evolution histories of popular IS in our dataset

System Name	System type	# of schema versions	lifetime (years)
ATutor	Educational CMS	216	5.7
CERN DQ2	Scientific DB	51	1.3
Dekiwiki	CRM, ERP	11	1.11
E107	CMS	16	5.4
Ensembl	Scientific DB	412	9.8
KT-DMS	CMS	105	4
Nucleus CMS	CMS	51	6.7
PHPWiki	Wiki	18	4.11
SlashCode (slashdot.org)	News Website	256	8.10
Tikiwiki	Wiki	99	0.9
Mediawiki (Wikipedia.org)	Wiki	242	6.2
Zabbix	Monitoring solution	196	8.3

speed up. The effect of this technique is included in the baseline performance in Section 5, since present in prior literature.

Another significant performance improvement is obtained by an extended version of the pruning technique appeared in [29]. We refer to 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 [29]. Our implementation can, in fact, also operate under 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 our experimental Section 5.

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.

D.2 Caching

Observing the workloads from Wikipedia, $Ensembl$ and the other information systems from Table 5 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 queries). 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.

D.3 Back and Forth from SQL

The last question that remains to be answered is how to translate back and forth between the SQL and query-equivalence-based representation of updates. 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, i.e., 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 Section 5.

E. EXPERIMENTAL SETTINGS

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. Table 5 reports the complete set of evolution histories that we used from [10].

E.1 Effect of foreign key on rewriting time

The results reported in Figure 5B are based on the following experiment. We tested with five simple queries (results for updates are derived 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 how the actual schema layout is not relevant to the rewriting performance, i.e., having N tables directly reachable with a single-hop from the query footprint or N tables reachable through a long chains 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 influence the size of the mapping, expressed as DEDs, that we feed into the chase engine MARS. Rewriting time are presented for both the scenario in which we use back-chase to improve the output query quality and the rewriting time when no query optimization is performed. Thanks to the nature of the backchase-based optimizer we utilize [14] 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.

E.2 Wikipedia Queries

The total number of query and update templates is typically rather small (less than a thousand for Wikipedia), therefore, the cache sub-

127

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.

stitution 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. This results are derived from the online profiler of Wikipedia `http://tinyurl.com/wikipediaprofiler`.

Below we report the 3 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):*

```
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;
```

Table 7: Schema Evolution Tools Comparison

		DB2 CM Expert	Oracle CM Pack	MySQL Workbench	IDERA SQL CM	Embarcardero CM	RedGate	DTM DB Suite	SwisSQL	Liquibase	PRISM++
Schema	Doc	✓	?	✓	✓	✓	✓	?	✓	✓	✓
	Predict	✓	✓	✗	✓	?	?	✓	✗	✗	✓
	Transform	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Reverse	✓	✓	✓	✓	✓	✓	?	✓	✓	✓
Data	Doc	?	?	✓	✗	✗	✓	?	✓	✓	✓
	Predict	✓	✓	?	✗	✗	?	✓	✗	✗	✓
	Transform	✓	✓	✓	✗	✗	✓	?	✓	✓	✓
	Reverse	?	?	✓	✗	✗	✓	?	?	✓	✓
Query	Predict	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
	Transform	✗	✗	✗	✗	✗	?	✗	✗	✗	✓
Update	Predict	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
	Transform	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
Indexes, Triggers Store Proc., etc.	Predict	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	Transform	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗

F. SOFTWARE TOOLS COMPARISON

In Table 7 we report a comparison with some of the most popular tools. 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), e.g., 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.