Managing and Querying Transaction-time Databases under Schema Evolution

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

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

Alin Deutsch
UC San Diego
deutsch@cs.ucsd.edu

Chien-Yi Hou
UC San Diego
cyhou@cs.ucsd.edu

Carlo Zaniolo
UCLA
zaniolo@cs.ucla.edu

ABSTRACT

The old problem of managing the history of database information is now made more urgent and complex by fast spreading web information systems, such as Wikipedia. Our $\mathcal{PRIMA}$ system addresses this difficult problem by introducing two key pieces of new technology. The first is a method for publishing the history of a relational database in XML, whereby the evolution of the schema and its underlying database are given a unified representation. This temporally grouped representation makes it easy to formulate sophisticated historical queries on any given schema version using standard XQuery. The second key piece of technology is that schema evolution is transparent to the user: she writes queries against the current schema while retrieving the data from one or more schema versions. The system then performs the labor-intensive and error-prone task of rewriting such queries into equivalent ones for the appropriate versions of the schema. This feature is particularly important for historical queries spanning over potentially hundreds of different schema versions and it is realized in $\mathcal{PRIMA}$ by (i) introducing Schema Modification Operators (SMOs) to represent the mappings between successive schema versions and (ii) an XML integrity constraint language (XIC) to efficiently rewrite the queries using the constraints established by the SMOs. The scalability of the approach has been tested against both synthetic data and real-world data from the Wikipedia DB schema evolution history.

1. INTRODUCTION

The ability of archiving past database information and supporting temporal queries over historical databases has long been recognized as highly demanded in Information Systems (IS) [19]. This objective, which has provided a long standing motivation for temporal database research, is now making significant inroads in the commercial world with the introduction of concepts such as 'flashback' and related constructs in the DBMS of major database vendors [1]. But even more telling than the slow movements among the DBMS pachyderms, there is the explosive spreading of collaborative web information systems such as Wikipedia. Due to their obvious accountability obligations about the past information divulged to the world, these systems preserve much of the information contained in their underlying databases, and they support versioning for their content (text and multimedia). In addition to content evolution [3], these systems experience intense evolution of database schema: as reported in [9], Wikipedia has experienced more than 170 schema changes in its 4.5 years of lifetime. Thus, schema evolution, which represents a serious problem for traditional information systems [24, 15], is even more critical for web information systems, particularly when it comes to preservation of the database history.

Converting the old database to a new one after each schema change might compromise the archival quality of the historical database, since information and relationships embedded in the original database are often lost, especially, those that are not well-understood or are considered unimportant by the DBA performing the conversion. In order to avoid this problem and to achieve perfect archival, it is required to store the data *as is*, under their original schema versions.

This important user requirement, however, introduces a serious challenge to the DBMS, namely managing and querying transaction-time databases with evolving schemas. To be more specific, the following two major issues need to be addressed: (i) effective and efficient archiving of transaction-time databases with evolving schemas and (ii) efficient query answering on temporal data under schema evolution. In our transaction-time database system $\mathcal{PRIMA}$[1], we address these problems as follows:

A unified representation for the history of databases and their schemas. Representing evolving data with evolving schema is inherently a difficult problem. Previously, two solutions have been proposed for such data representation: single-pool and multi-pool solutions [7]. These solutions, however, are based on the relational model, which is essentially too restrictive for managing evolving data with evolving schemas. Instead, we propose to uniformly represent the history of both databases and their schemas in an XML document, called Multiversion V-document, or MV-document. MV-documents use a temporally grouped data model [8] that naturally and concisely represents temporal data and also simplify the task of expressing powerful temporal queries using standard XQuery. In [27], it also has been shown that advanced RDBMS technology can be

[1]$\mathcal{PRIMA}$ stands for Panta Rhei Information Management & Archival

Table 1: Schema evolution in an employee database (V_1 to V_5)

	Schema Versions	T_s	T_e	SMO Sequence
V_1	engineerpersonnel (empno, name, hiredate, title, deptname) otherpersonnel (empno, name, hiredate, title, deptname) job (title, salary)	T_1	T_2	`MERGE TABLE engineerpersonnel, otherpersonnel INTO empacct;`
V_2	empacct (empno, name, hiredate, title, deptname) job (title, salary)	T_2	T_3	`CREATE TABLE dept;` `COPY COLUMN deptname FROM empacct INTO dept;` `ADD COLUMN deptno AS gen_id(deptname) INTO dept;` `COPY COLUMN deptno FROM dept INTO empacct WHERE empacct.deptname = dept.deptname;` `DROP COLUMN deptname FROM empacct;` `ADD COLUMN managerno INTO dept;`
V_3	empacct (empno, name, hiredate, title, deptno) job (title, salary) dept (deptname, deptno, managerno)	T_3	T_4	`CREATE TABLE empbio;` `COPY COLUMN empno FROM empacct INTO empbio;` `ADD COLUMN sex into empbio;` `ADD COLUMN birthdate into empbio;` `COPY COLUMN name FROM empacct INTO empbio WHERE empbio.empno = empacct.empno;` `DROP COLUMN name FROM empacct;`
V_4	empacct (empno, hiredate, title, deptno) job (title, salary) dept (deptname, deptno, managerno) empbio (empno, sex, birthdate, name)	T_4	T_5	`ADD COLUMN firstname AS getFirstName(name) INTO empbio;` `ADD COLUMN lastname AS getLastName(name) INTO empbio;` `DROP COLUMN name FROM empbio;` `COPY COLUMN salary FROM job INTO empacct WHERE empacct.title = job.title;` `DROP TABLE job;`
V_5	empacct (empno, hiredate, title, deptno, salary) dept (deptname, deptno, managerno) empbio (empno, sex, birthdate, firstname, lastname)	T_5	*now*	

exploited for efficient query execution on this XML-based transaction-time data. We discuss this in Section 2.

User-friendly Interface for Querying. To retrieve the contents of the database at a time T, a user can first identify in the MV-document the tables (and their columns) valid at time T, and then write a suitable query on them. While walk back in history is supported by $\mathcal{PRIMA}$ user interface, which might be invaluable for a new DBA learning the system, it can be too taxing for casual users who want to ask simple queries such as: "What was Joe Doe's salary on January 1, 1997?" Even worse, a query such as "What is Joe Doe's salary from January 1, 1997, to date?" would require a different query on each schema version that was valid since January 1, 1997.

Therefore, $\mathcal{PRIMA}$'s unique solution consists in letting a user express her query based only on the current schema version (or any other schema version of her choice). Then $\mathcal{PRIMA}$ translates this input query into equivalent queries against all applicable schema versions and executes them against the databases underlying such schemas. Unless the user explicitly asks for the translated queries and their supporting schemas, this process is totally transparent: $\mathcal{PRIMA}$ returns the same result as if i) the old data had first been converted to the current schema, and then ii) the input query had been executed on the data so converted[2]. In previous works, it was proposed to literally implement the semantics, to migrate data and then answer the input queries [20]. This, however, is prohibitively expensive in most of the serious transaction-time databases where historical data ever increases in size. $\mathcal{PRIMA}$ achieves the same semantics in an efficient manner, by means of query reformulation.

Taxonomy-based Query Reformulation. Query reformulation, recently, has been studied in the context of snapshot data with a single source version [12, 29]. We, however, address a different and more challenging problem of rewriting temporal queries into equivalent ones that will be evaluated against one or more source versions of evolving data. To address this problem, we propose a novel solution capable of finding correct and efficient rewritten queries based on a query taxonomy: we characterize an input temporal query, based on the predefined criteria and the query analysis algorithms. (Section 3) We then exploit the characteristics of the input query in producing a correct and efficient rewritten query, as discussed in Section 4.

Optimization. Even though we ensure that the rewritten query is correct and efficient, the reformulation algorithm itself needs to be efficient also. This turns out to be another serious challenge, as the real-world data, such as Wikipedia schema evolution, insist that we need to handle hundreds of schema versions [9]. In order to perform query rewriting between two schema versions that are hundreds of versions away, within a practical time bound, we propose two optimization techniques in Section 5.

Experimental Study. We validate the proposed approach using both synthetic and real world data, in terms of usability, correctness and efficiency of rewritten queries, and efficiency of reformulation algorithm. Especially, real data of Wikipedia, which represent 171 schema versions over 4.5 year period, prove the practical effectiveness of our approach. (Section 6)

1.1 Motivating Example

To better motivate the need for schema evolution support, we illustrate a schema evolution example in an employee database, which is used as a running example in the rest of the paper[3]. Table 1 outlines our example, which has five schema versions, V_1 through V_5. It also shows the schema modification operators (SMOs), which are discussed in detail in Section 2.1.

Since the establishment of the database at T_1, schema version V_1 initially had three tables: **engineerpersonnel**, **otherpersonnel** and **job**. The first two store information about the engineers and the rest of the personnel, respec-

[2]This feature is called *schema versioning* [22].

[3]The example is borrowed from [23] with some adaptation and addition.

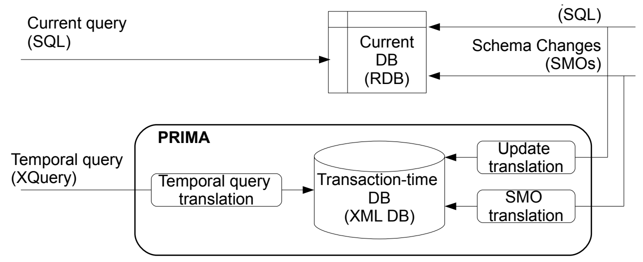


Figure 1: PRIMA Architecture

tively. The table **job** relates the employee job titles to the corresponding salaries. Now, due to the changes in business requirements and operating environment, the schema receives the following sequence of modifications.

As the company seeks to uniformly manage the department information, the DBA applies the first modification at time T_2, which merges two tables **engineerpersonnel** and **otherpersonnel**. We obtain the schema of V_2, as a result.

With the growth of the company, the need for storing more information about departments leads to a new schema version V_3, which occurred at time T_3. In V_3, a new table **dept** was created, which stores department number, department name, and the manager[4], for each department.

After a while, due to a new government regulation, the company is now required to store more personal information about employees. At the same time, it was required to separate employees' personal profiles from their business-related information to ensure the privacy. For these reasons, the database layout was changed at T_4, to the one in version V_4, where the information about the employee is enriched and divided into two tables: **empbio**, storing the personal information about the employee, and **empacct**, maintaining business-related information about the employee.

Finally, the company chose to change its compensation policy: to achieve 'fair' compensation and to better motivate employees, the salary is made dependent on the individual performance, rather than on his or her job title. To support this, the salary attribute was moved to the **empacct** table, and the table **job** was dropped. Another modification was also introduced, to simplify the surname-based sorting of employees: thus, the employee first name and last name are now stored in two different columns. These changes, which were applied at T_5 introduced the last schema version, V_5.

2. PRIMA ARCHITECTURE

Figure 1 shows the high-level architecture of PRIMA system: given a current or snapshot database in relational model, we construct its historical database or transaction-time database. Transaction-time DB is constructed and evolved using the data changes (SQL insert/update/delete) and schema changes (SMO) posed on the snapshot DB, with help of the translation modules, these changes are automatically propagated to the transaction-time database. With the transaction-time DB, users can ask temporal queries in XQuery, when they are interested in the history of the database, while the regular application queries on the current database will continue to use the same relational database, without interruption.

[4]We assume that the existing **empacct.title** in V_2 does not contain manager information, which is newly introduced.

Table 2: Schema Modification Operators (SMOs)

SMO
CREATE TABLE t;
DROP TABLE t;
RENAME TABLE t INTO t';
DISTRIBUTE TABLE t INTO t_1 WITH cond, t_2 WITH cond;
MERGE TABLE t_1, t_2 INTO t;
ADD COLUMN c AS $f(c_1, c_2, ..., c_n)$ INTO t;
DROP COLUMN c FROM t;
RENAME COLUMN c IN t TO c';
COPY COLUMN c FROM t_1 INTO t_2 WHERE cond;
MOVE COLUMN c FROM t_1 INTO t_2 WHERE cond;

In the following subsections, we first discuss the types and syntax of the schema changes that we allow and then present our XML-based transaction-time database (MV-docment), in greater detail.

2.1 Schema Modification Operators

Schema modification operators (SMOs) are a set of operators capable of describing schema evolution. These have been used to describe the schema evolution in our running employee example in Table 1. We summarize SMOs supported in $\mathcal{PRIMA}$ in Table 2, which shows five table-level SMOs (first five) and five column-level SMOs (the rest).

SMOs take a schema version as an input and produce a new schema version. The first three operators in Table 2 are for table creation, destruction, and renaming, which are taken from the SQL:2003 standard. DISTRIBUTE TABLE and MERGE TABLE are the two table-level SMOs, for horizontal table division and table merge.

Rest of the SMOs specify column-level schema modification. The first three constructs are taken from the SQL standards: ADD COLUMN introduces a new column, where the new column can be filled by a user defined function or constant (*NULL* by default). DROP COLUMN removes an existing column from a table, deleting all data in the column. RENAME COLUMN changes the name of a column, without affecting the data. The last two SMOs are COPY COLUMN and MOVE COLUMN. With these two operators, users can express arbitrarily complex structural changes in relational model, including join, decomposition, and normalization. Thus, COPY COLUMN makes a copy of a column A from a table $T1$ to another table $T2$, by (i) taking a join of table $T1$ and $T2$, (2) projecting $T2 \cup \{A\}$ from the join result, and (3) replacing $T2$ with the projection. As a special case, if $T2$ has no existing columns, then WHERE clause is not required and $T2$ simply gets a new column A, a copy of all values in $T1.A$. MOVE COLUMN is a shorthand for a COPY COLUMN followed by DROP COLUMN on the first table.

By combining one or more of these SMOs in a sequence, a new schema version is introduced, as shown in Table 1. It has been shown that complex schema evolution scenario in Wikipedia schema's 171 versions, can be completely and naturally described using these operators [9]. As shown in the following sections, SMOs are also supportive of efficient storage of historical data under schema evolution (Section 2.2.3) and efficient query reformulation (Section 5).

2.2 XML-Based Transaction-time Databases

The problem of preserving and querying the history of a database has long been studied in temporal database research community [19]. While research efforts in 90's demon-

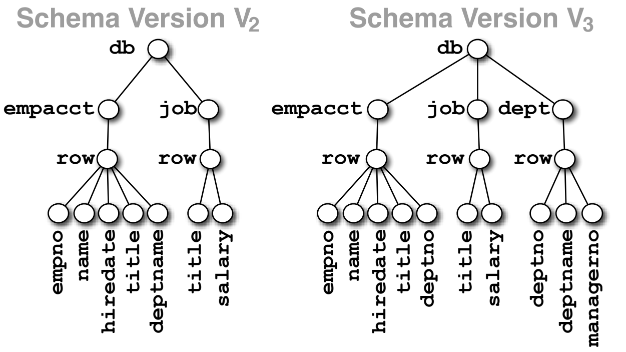


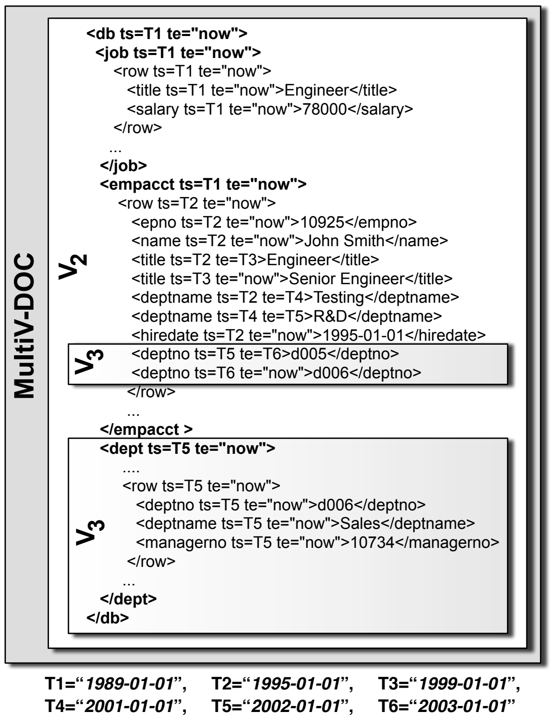
Figure 2: Employee DB Schema Versions (XML tree representation for versions V_2 and V_3)

strated the difficulty of devising simple temporal extensions to the relational data model, the temporal difficulties of the relational model and SQL dissolve when the database information is viewed and queried using XML and its query languages [21, 27]. In particular, in [27] it was shown that the history of a relational database can be published in XML, and viewed under a temporally grouped representation whereby complex historical queries can be expressed in standard XQuery. These temporal queries and views can be managed in any DBMS that provides native support for XML and XQuery, or more efficiently managed in a Relational DBMS that supports the SQL/XML standards as in [27]. We describe this XML-based temporal model, called V-document, and show how we extend it to model and query temporal databases with evolving schemas.

2.2.1 V-Document

In Figure 2, an example V-document is shown for schema V_2 and V_3 of our running example. Under the ***db*** root, we find the tables for the current schema. Under each table, we have a set of ***row*** elements, where each element represents the history of a row in the table. At the last level is a history of each column value in a row, named by the column name. Using XPath-like notation, the structure of V-document can be represented as: ***/db/table-name/row/column-name***. Each such element is timestamped with its period of continuous existence by its **ts** (start time) and **te** (end time) attributes, where the data is valid on **ts** (inclusive), but not on **te** (exclusive). The only exception to this is where we use a special value "now" for the end time, which means that the associated value is currently valid. Moreover, there is a hierarchical relationship between the timestamp periods, whereby the period of the ***db*** contain the periods of the tables, which in turn contain the periods of the ***row*** elements; these in turn contain those of the column-name elements under them.

XQuery can be used as an effective temporal query language, over this representation, without requiring extensions to the standards [27]. This is due to (i) the expressive power of XQuery which is Turing-complete [14], and (ii) the fact that by supporting a temporally grouped representation XML reduces the need for coalescing [8]. We next present examples of temporal queries on our employee database. For these queries, we assume that the transaction-time DB is stored as a V-document, with a single schema version V_3.



MultiV-DOC
V_2
V_3
V_3

```
<db ts=T1 te="now">
 <job ts=T1 te="now">
   <row ts=T1 te="now">
     <title ts=T1 te="now">Engineer</title>
     <salary ts=T1 te="now">78000</salary>
   </row>
   ...
 </job>
 <empacct ts=T1 te="now">
   <row ts=T2 te="now">
     <epno ts=T2 te="now">10925</empno>
     <name ts=T2 te="now">John Smith</name>
     <title ts=T2 te=T3>Engineer</title>
     <title ts=T3 te="now">Senior Engineer</title>
     <deptname ts=T2 te=T4>Testing</deptname>
     <deptname ts=T4 te=T5>R&D</deptname>
     <hiredate ts=T2 te="now">1995-01-01</hiredate>
     <deptno ts=T5 te=T6>d005</deptno>
     <deptno ts=T6 te="now">d006</deptno>
   </row>
   ...
 </empacct >
 <dept ts=T5 te="now">
   ....
   <row ts=T5 te="now">
     <deptno ts=T5 te="now">d006</deptno>
     <deptname ts=T5 te="now">Sales</deptname>
     <managerno ts=T5 te="now">10734</managerno>
   </row>
   ...
 </dept>
</db>
```

T1="*1989-01-01*", T2="*1995-01-01*", T3="*1999-01-01*",
T4="*2001-01-01*", T5="*2002-01-01*", T6="*2003-01-01*"

Figure 3: Employee DB example: MV-Document

Q1. Find the number of employees working in the R&D department on 1992-01-01.

```
for $db in doc("empdb.xml")/db,
    $r1 in $db/empacct/row,
    $r2 in $db/dept/row,
    $c1 in $r1/deptno,
    $c2 in $r2/deptno
where $r2/deptname="R&D" AND $c1=$c2 AND
      $c1/@ts<="1992-01-01" AND $c1/@te>"1992-01-01"
return count($r1);
```

Q2. Find all department names in history for the employee 10925.

```
for $db in doc("empdb.xml")/db,
    $r1 in $db/empacct/row,
    $r2 in $db/dept/row
where $r1/empno="10925" AND $r1/deptno=$r2/deptno
return $r2/deptname;
```

2.2.2 V-Document with Evolving Schemas

In order to represent the history of a relational database where the schema evolves along with the content, we naturally extend V-document. For instance, assume that there is a major change in the schema, such that we move from the two-table schema version V_2 to a three-table schema version V_3 shown in Figure 2. This change is represented in XML by simply appending the names of the columns and tables after the old ones as shown in Figure 3. The timestamp values associated with tables and tuples unambiguously identify which schema a tables belongs to, and which table each tuple belongs to. This is highlighted in Figure 3 with boxes.

Thus we have a general representation, named MV-Document, capable of representing both the content and the history of our database using a standard XML representation. Note that all historical data are stored under their original schema version, satisfying our archival requirement. However, querying such a database can become quite complex. For instance, for a snapshot query at time t we will have to first find which subschema is valid at time t and formulate the query accordingly. Thus, queries posed on different schema version may need to be rewritten to fit the correct schema, e.g., query Q1 posed in terms of schema V_3 should be rewritten in terms of schema V_2 (since it refers to data prior to the schema evolution). Furthermore, for a history query that spans a certain period of time, the user will have to write one query for each schema version valid in the period. In the worst case, the user will have to revisit and understand the list of all past schemas, a list that is bound to grow as the years go by.

Thus, in $\mathcal{PRIMA}$, we seek a much easier-to-use solution: we propose automatic rewriting of input queries that are expressed on a schema version into equivalent queries against one or more applicable source schema versions. For example, $\mathcal{PRIMA}$ rewrites queries Q1 and Q2 as follows:

Q1'. Find the number of employees working in the R&D department on 1992-01-01. (written against V_2, the only source version of Q1)

```
for $db in doc("empdb.xml")/db,
    $r1 in $db/empacct/row,
    $c1 in $r1/deptname
where $c1= "R&D" AND $c1/@ts <="1992-01-01" AND
      $c1/@te>"1992-01-01"
return count($r1);
```

Q2'. Find all department names in history for the employee 10925. (written against all five source versions)

```
for $db in doc("empdb.xml")/db,
    $r1 in $db/empacct/row,
    $r2 in $db/dept/row,
where $r1/empno="10925" AND $r1/deptno=$r2/deptno
return $r2/deptname
union
for $db in doc("empdb.xml")/db,
    $r1 in $db/empacct/row,
where $r1/empno="10925"
return $r1/deptname;
```

Note that the first subquery of $Q2'$ queries V_1 and V_2, and the second one queries V_3, V_4, and V_5.

Schema Version Each schema version V_i $(1 \leq i \leq N)$ is valid for its validity period P_i, [start-time, end-time). Note that the last schema version V_N has *now* as its end time of validity period, which means that it is valid up to now (inclusive) and it will continue to be valid until a new schema version is introduced. When given an input query, we call the schema version against which the query is written as *target version*, and the schema versions that contain the data that may contribute to the query answers as *source versions*.

2.2.3 Storage Partitioning at Schema Changes

Given an MV-document, when schema changes are introduced there are several options on temporal data storage method. Among others, we consider the following two: the first is, to *break* all tables, at any schema change. Breaking a table, consists of i) terminating the existing table by setting all end times in the table that are equal to "now", to the current timestamp at the schema change, and ii) starting a new table for those terminated rows and tuples by the schema change. Another storage option is to break a table only if the table was affected by the schema change. We call the former as *global break* and the latter as *local break*.

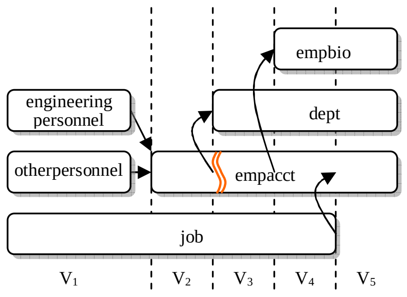


Figure 4: Partitioning of Employee Database

Global break is simpler and more intuitive as all tables are broken at all schema changes. It, however, incurs significant storage overhead: often, schema changes are local to parts of schemas and it brings unnecessary storage overhead to break majority of the tables that are not affected by SMOs. We therefore choose to use local break for our MV-document, where the tables are broken only when it's necessary.

For local break, we again have two following options: we may break tables i) at all SMOs, or ii) at the table-level SMOs only. Since fewer break implies the less storage overhead, we choose the second option. For example, as shown in Table 1.1, when **deptname** was dropped from **empacct** and **deptno** was added into **empacct** between V_2 and V_3, we do not need to break the table, but can instead set the end times of all **deptname** column values as 2003-01-01 and start a new column **deptno**.

The table break that we discussed above, is also called *hard break* and for a table, all versions between two hard breaks are called a *hard partition*. Given that, we discuss another type of break, called *soft break*, which creates a *soft partition*. When there is a column-level SMO, the affected tables do not need to be broken. However, as column-level SMO affects the schema, we need two different queries to access the data before and after the column-level SMO. We note that all SMOs cause a soft break, except for DROP COLUMN: dropping a column does not require a query valid on the post-SMO version to be rewritten on the pre-SMO version and the same query can be run on the pre-SMO version.

EXAMPLE 2.1. Using this storage scheme, Table 1 stored as MV-document has six table-level elements, namely ***engineerpersonnel***, ***otherpersonnel***, ***job***, ***empacct***, ***dept***, ***empbio***. This is shown in Figure 4, which represents hard partitions with six rounded-rectangles. The figure also shows two soft partitions in **empacct**, around a curly line that indicates the only soft break in employee database example. This soft break is caused by COPY COLUMN smo between V_2 and V_3 that copies DEPTNO from DEPT table to EMPACCT table. (see Table 1) □

3. TEMPORAL QUERY TAXONOMY

Reformulating a temporal query into multiple source versions is a nontrivial task. In order to produce a correct and efficient rewritten queries, we characterize an input query, based on our temporal query taxonomy, which is presented in this section.

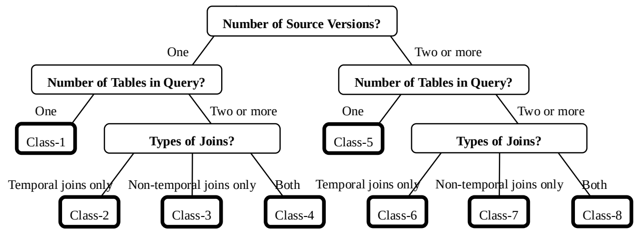


Figure 5: Temporal Query Taxonomy

3.1 Criteria and Query Classes

We use three criteria to categorize input queries into eight different query classes, as shown in Figure 5. The first criterion tests whether the query has only a single source version or two or more source versions. Recall that source versions of an input query Q are all schema versions where Q can find its answers.

Example 3.1. Let us consider the schema versions in Table 1 whose two schema versions have the following validity periods: V_1 between [2001-01-01, 2002-01-01), and V_2 between [2002-01-01, 2003-01-01). If a user input query Q_1, written on V_3, asks to retrieve the salary value of Joe Doe at 2001-07-01, the query Q_1 has only one source version, namely V_1. Similarly, if Q_2 written on V_3 asks to retrieve all salary values of Joe Doe during [2001-01-01, 2001-07-01), this also has only one source version, V_1. If another query Q_3 asks to retrieve the same information as in Q_2, but during [2001-12-01, 2002-02-01), then Q_2 is a multiple source version query, with two source versions, V_1 and V_2. □

In general, it is cumbersome to ask users for source versions of input queries as they may not precisely know the schema version history. Instead, we analyze the input query to find out the source version of a given query, which is done by our *MinSourceFind* algorithm in Section 3.2.1.

The second criterion checks how many tables a query Q accesses[5] and tells whether it has only one table or two or more tables. This can be easily tested, by parsing the input query. One thing to note is, for our purpose, we count multiple accesses to the single table as multiple accesses. For example, if a query has a self-join, then it accesses two tables, we say. The reason becomes clear in the next criterion.

The last criterion checks the types of joins between the queried tables, when a query accesses two or more tables. We see whether the joins in the given query are i) all temporal joins, ii) all non-temporal joins, or iii) both. Temporal join is a join where two tuples from two tables are required to temporally overlap and to have a non-empty intersection of validity periods. We present an algorithm called *TJoinFind*, which analyzes the types of joins in the given query.

Example 3.2. Let us assume that a user writes a query Q_4 against V_3, asking for the employees who worked in the department where Joe Doe worked in, with an overlapping period. This query then has a temporal join between **empacct** and **empacct**. If we remove the phrase "with overlapping period" from Q_4's description, it becomes a query with non-temporal join. □

[5] *Accessed tables* mean the tables in queries' FROM clause, in case of SQL query, and the number of table-level elements used, in case of MV-Document XQueries.

Using these three criteria, we classify queries into eight classes, as shown in Figure 5. In the next subsection, we present how the two criteria in this subsection, except the second one, can be tested.

3.2 Query Analysis Algorithms

3.2.1 MinSourceFind

We present an algorithm *MinSourceFind*, which finds the minimal set of source versions, given an input query. The algorithm exploits the temporal conditions in the input query to prune the hard partitions and soft partitions as in the following two phases. For the discussion, we use $\text{HP}_{\ni T}$ ($\text{SP}_{\ni T}$) to denote the hard partition (the soft partition) with a validity period that contains the timestamp T.

Phase 1: Hard Partition Pruning We begin by assigning all possible hard partitions to each of element nodes that a given query accesses. We then repeatedly apply the pruning rules A1 through B3 in Table 3, which shows how we prune, given a temporal condition on an element node. We illustrate this using the following example, explaining why they are safe rules.

Example 3.3. Consider the following query Q3 posed on the employee database V_5, which retrieves all employees who were working on 1997-07-01.

```
for $c in doc("empdb.xml")/db/empbio/row/empno
where $c/@ts<"1997-07-01" AND $c/@te>"1997-07-01"
return $c;
```

With this query, let us examine how its hard partitions are pruned. We assume that 1997-07-01 falls within V_2's validity period. In the beginning, we determine that `$c` has three hard partitions, $\text{HP}_1=\{V_1\}$, $\text{HP}_2=\{V_2, V_3\}$, $\text{HP}_3=\{V_4, V_5\}$ (see Figure 4), as `$c` is a column of the table **empbio**. Based on the condition `$c/@ts<"1997-07-01"`, we apply the rule A1 in Table 3 with $\text{HP}_{\ni 1997\text{-}01\text{-}01}=\text{HP}_2$, and can prune away $\text{HP}_3=\{V_4, V_5\}$ from `$c`'s hard partitions. This is a safe pruning, because all data items in HP_3 have its `@ts` greater than 1997-07-01, and would never satisfy the given condition. Similarly, using `$c/@te>"1997-07-01"` and the rule A3 in Table 3, we prune out $\text{HP}_1=\{V_1\}$, leaving $\text{HP}_2=\{V_2, V_3\}$ as `$c`'s only hard partition to query. □

Pruning rules B1, B2, and B3 are similar to A1, A2, and A3, except that the pruning is based on the hard partitions of another element, rather than $\text{HP}_{\ni T}$. For example, rule B2 shows that, if we are given a query condition that an element e has a start time equal to start time or end time of another element e_2, then we can prune some hard partitions from e based on the already-pruned hard partitions of e_2: if e has a certain hard partition that does not overlap with any of e_2's hard partitions, then we can safely prune it because it cannot have any tuple with a start time equal to e_2's start time or end time. Similarly, we establish B1 and B3 in Table 3. Note that one temporal condition between e and e_2 (e.g. $e < e_2$) can be used twice for pruning by changing the roles of e and e_2 (e.g. $e_2 > e$).

When we have a conjunctive or a disjunctive of temporal conditions for an element node, we compute its hard partitions as follows. Conjunction (AND) of temporal conditions for an element node, has an intersection of two sets of hard

Table 3: Storage Partition Pruning Rules based on Input Query Conditions

Rule	Query Condition	Partitions to Prune
A1	$e/@ts < T$ or $e/@te < T$	e's HPs after HP$_{\ni T}$
A2	$e/@ts = T$ or $e/@te = T$	e's HPs except HP$_{\ni T}$
A3	$e/@ts > T$ or $e/@te > T$	e's HPs before HP$_{\ni T}$
B1	$e/@ts < e_2/(@ts\|@te)$ or $e/@te < e_2/(@ts\|@te)$	e's HPs after the last HP that overlaps with e_2's HPs
B2	$e/@ts = e_2/(@ts\|@te)$ or $e/@te = e_2/(@ts\|@te)$	e's HPs that do not overlap with any of e_2's HPs
B3	$e/@ts > e_2/(@ts\|@te)$ or $e/@te > e_2/(@ts\|@te)$	e's HPs before the first HP that overlaps with e_2's HPs
C1	$e/@ts = T$ or $e/@ts > T$	e's SPs before SP$_{\ni T}$
C2	$e/@te < T$ or $e/@te = T$	e's SPs after SP$_{\ni T}$
D1	$e/@ts = e_2/(@ts\|@te)$ or $e/@ts > e_2/(@ts\|@te)$	e's SPs before the first SP that overlaps with e_2's SPs
D2	$e/@te < e_2/(@ts\|@te)$ or $e/@te = e_2/(@ts\|@te)$	e's SPs after the last SP that overlaps with e_2's SPs
E1	$e/@ts < T$ or $e/@ts = T$	e's SPs after SP$_{\ni T}$, if e has SP$_{\ni T}$
E2	$e/@te = T$ or $e/@te > T$	e's SPs before SP$_{\ni T}$, if e has SP$_{\ni T}$

partitions from the two conditions, as its set of hard partitions, while disjunction (OR) has a union of two sets.

After finding hard partitions for each element node at all levels, we merge them as follows, to come up with the hard partitions of table-level element nodes: parent element node[6] has an intersection of all children element nodes' hard partitions and its own hard partitions, as its new set of hard partitions. Under a table-level element, all element nodes are temporally bound within a hard partition, so if one of the element nodes has a condition that cannot be satisfied in a hard partition, we do not need to access the table in that particular partition and can safely prune that hard partition. After this, we assign table-level hard partitions to all element nodes under the table-level element. From these hard partitions of each element node in a query, we get all soft partitions that will be further pruned in Phase 2.

Phase 2: Soft Partition Pruning Sometimes, it may not be possible to prune a hard partition as a whole, but it might be possible to prune some soft partitions contained in it. Pruning soft partition, however, requires different rules because data are not broken into two different tables in the storage, at the soft breaks. Rules C1 through E2 in Table 3 show those rules. Rule C1, as an example, tells that we can safely prune a soft partition, if it precedes SP$_{\ni T}$, because no element data in that soft partition will satisfy the condition. As in the case of hard partition pruning, D1 and D2 can be applied together with C1 and C2 to further prune soft partitions using other element nodes' already-pruned soft partitions.

Rules E1 and E2 also prune soft partitions, but the reason that they work is a bit different from that of C1,C2,D1, and D2. We explain this in the following example.

EXAMPLE 3.4. We continue to discuss the Q3 in Example 3.3. From the only hard partition left to `$c`, we obtain two soft partitions contained in it, namely SP$_1$={V_2} and SP$_2$={V_3} (see Figure 4). By applying rule E1 with the condition `$c/@ts<"1997-07-01"` and SP$_{\ni 1997\text{-}01\text{-}01}$=SP$_1$, we prune away SP$_3$. This pruning is safe, since all data item that satisfy the given condition must be spanning (or valid on) SP$_1$, and thus querying SP$_2$ is not necessary. Therefore, after pruning we return SP$_1$={V_2}, as the only soft partition, and also the only source version of Q3. □

Note that rules E1 and E2 are used only after D1 and D2 are applied, since applying D1 and D2 after E1 and E2 produces unsafe pruning: all pruning rules from A1 through D2 prune partitions when they don't produce answers for the query, but E1 and E2 prune soft partitions where they have answers.

After these two phases, we merge all soft partitions of all element nodes, by unioning them and breaking whenever there is a soft break. We return one source version from each soft partition. If we have only one source version, it means that the given query is among Class-1,2,3, or 4. If it has two or more source versions, we still use the minimal source versions found for optimal query reformulation.

3.2.2 *TJoinFind*

Now we present an algorithm that tests whether two tables in a query are joined by temporal join or not. We first introduce terminologies used in discussion as follows:

When an input query specifies that two of its element nodes have overlapping intervals, we say that these two element nodes are *temporally overlapped*. Also, for a set of element nodes, if any pair of two element nodes in the set are temporally overlapped, we call the set as a *temporally overlapped node set* or *TONS*. When we know that all table-level element nodes in an input query are in a single temporally overlapped node set, then we say that all joins are temporal joins. We derive the following theorem on temporally overlapped node set, which is used in our testing algorithm.

THEOREM 1. *If a node N belongs to a temporally overlapped node set S, and P is a parent node of N, then S'=$S\cup\{P\}$ is also a temporally overlapped node set.*

PROOF. For each node N' in S, N has a non-empty overlap period $P_{N,N'}$ with N'. Also, due to the V-Document's temporal covering constraint, parent P's period contains child N's period. P's period hence also contains $P_{N,N'}$, and P is temporally overlapped with N', for all N' in S. Therefore, $S' = S \cup \{P\}$ is a temporally overlapped node set. □

Based on this theorem, we propose the following temporal-overlap testing algorithm:

- Step 1: find the explicit temporal overlap constraints, $e_1/@ts < e_2/@te$ AND $e_2/@ts < e_1/@te$ Using this, establish a TONS of two elements, S = {e_1, e_2}.
- Step 2: extend TONS with the element parents, based on the Theorem 1. Go to Step 2 if S was extended.
- Step 3: using the maximally-extended TONS, determine and report the pairs of tables in the input query that are temporally overlapped.

[6]Recall that the hierarchy of parent-child relationship of node elements is as follows: DB-level>table-level>row-level>column-level.

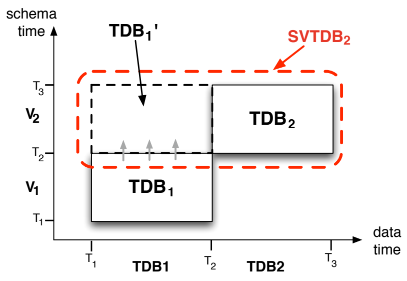


Figure 6: Transaction-time DB under V_1 and V_2

4. QUERY REFORMULATION

In this section, we present how we produce correct and also efficient rewritten queries for each query class.

4.1 Single Source Version Queries

In this subsection, we show how we reformulate all single source version queries, namely the queries in Class-1,2,3 and 4. Given that the input query Q has only one source version, we reformulate it into Q' that accesses the source version. For such reformulation, we discuss i) the semantics of query reformulation, with which we define the correctness of query reformulation, ii) how we generate schema mappings from the user-provided SMOs, and iii) how we reformulate queries using such generated mappings. Note that the reformulation mechanisms discussed in this section are exploited as a building block for reformulation of all other query classes with extensions.

4.1.1 Semantics

We have database history under multiple schema versions where we let the user query the database history as if it were under a single schema version that she queried, or the target version. As in [29], we define the semantics of query answering as following: given a target version V_k, we migrate each database D_i under V_i (i $\leq$ k) to V_k according to the schema mappings and obtain $D_{i\to k}$ valid on V_k. We, then, evaluate the input query on the union of $D_{1\to k}$, $D_{2\to k}$, ..., $D_{(k-1)\to k}$, D_k, called *single version transaction-time database* or *SVTDB*, which is the entire history of the transaction-time database translated into V_k to conform to the target schema version using the SMOs between the two versions. We name *SVTDB* under V_k as $SVTDB_k$. Figure 6 shows a case where k=2. When users pose a query Q on V_i, they assume that it is executed on $SVTDB_i$. Achieving this without materializing $SVTDB_i$ is the goal of query reformulation.

4.1.2 Transaction-time DB Schema Mapping

In our approach, DBAs describe schema evolution of the snapshot DB, using SMOs. We then translate these SMOs into XICs which we use for query reformulation

XML Integrity Constraint Within our transaction-time database, which is modeled in XML, we employ XML Integrity Constraints (XICs) [12] as our mapping language. XIC is a language for expressing intra- or inter-schema integrity constraints in XML, using a relational representation. It is similar to first-order logic, except that it uses XPath expressions in predicate atoms. The predicates can be binary, of the form [p](x, y), which is satisfied whenever y belongs to the set of nodes reachable from node x along the path p. Alternatively, predicates can also be unary, of form [p](y), which is satisfied if y can be reached from the root by following p. Two XICs are given below as examples: (1) says that all rows of **empacct** have **empno** and (2) represents a key constraint of **empacct.empno**.

$$\forall p \quad [//\text{empacct}/\text{row}](p) \rightarrow \exists q\ [./\text{empno}](p,q) \qquad (1)$$

$$\forall p,q,r \quad [//\text{empacct}/\text{row}](p) \wedge [./\text{empno}](p,q) \wedge [//\text{empacct}/\text{row}](r) \wedge [./\text{empno}](r,q) \rightarrow p=r \qquad (2)$$

Using XIC, we generate transaction-time database schema mapping using the following semantics: given an SMO, we 1) slice a tuple's history into a set of snapshot data at each time instance, 2) execute SMOs to translate each of these snapshot data, and 3) coalesce the translated snapshot data into historical data. Rather than directly executing this semantics, which is prohibitively expensive in general, we directly translate temporal data under one schema version into those that conform to another schema version. For example, if a table was renamed between two versions, we take the transaction-time data under the old version with an old table name, and map them to transaction-time data under the new version with a new name, which is shown in the following:

$$(RT_f) \quad [/v1db/R](x_1), [./@ts](x_1,s), [./@te](x_1,e), [./row](x_1,x_2) \rightarrow \exists y_1 [/v2db/S](y_1), [./@ts](y_1,s), [./@te](y_1,e), [./row](y_1,x_2)$$

$$(RT_b) \quad [/v2db/S](y_1), [./@ts](y_1,s), [./@te](y_1,e), [./row](y_1,y_2) \rightarrow \exists x_1 [/v1db/R](x_1), [./@ts](x_1,s), [./@te](x_1,e), [./row](x_1,y_2)$$

In forward direction (RT_f), we map a table-level element **R** in V_1 to **S** in V_2 such that they have identical table-level timestamps (i.e. s and e) and identical rows (i.e. x_2). To obtain a backward direction XIC (RT_b), we simply flip the body and the head of the forward XIC, which implies that no other source contribute to **S** in the new schema version. Note that the forward and the backward XICs together establish two-way inclusion dependencies, establishing an equivalence between two tables. We generate XIC mappings for other SMOs in a similar way, as shown in the Appendix.

4.1.3 Reformulation Algorithm

For query reformulation, we use a traditional relational database technique called *chase* [2], with which an input query is expanded to another equivalent query under the given set of rules. For chase execution, rather than reinventing the wheel, we adopt an efficient chase engine called MARS [12]. With chase at the core, we reformulate the input query Q in three stages: chase preparation, chase, and output XQuery reconstruction.

Chase Preparation We first translate SMOs into XICs as discussed in the previous subsection. Given an input query Q, we then translate the navigation part[7] of Q into XBind query, the relational query language that is essentially a conjunctive query with an extension of special atom for XML navigation, as in XIC. Lastly, we provide MARS with the target version where query is posed and the source versions to rewrite the query. Target version comes from the

[7]Note that XQuery is composed of navigation part and tagging template [12]

user, while the source version is provided by *MinSourceFind* algorithm.

Chase With the input of XBind query, XIC, and a source version to rewrite to, we perform a chase, to rewrite an input query on the target version to an equivalent query on the source version. To be more specific, we use MARS to chase the query into the universal plan, the largest equivalent query, and then removes the atoms that are not in the source version. After the removal, we make sure that the reduced query can be chased back to the original query, ensuring their equivalence. Note that this holds in all cases in $\mathcal{PRIMA}$ as we design forward and backward XIC rules of an SMO to ensure this.

Output XQuery Construction In this stage, we use the rewritten XBind query obtained from the chase stage and also the input XQuery. We simply replace the navigation part of the input XQuery, using the rewritten XBind query while the tagging template remains the same.

We now have discussed how a single source version query is rewritten based on MARS. Now we discuss how we extend this for each query class.

4.2 Class-5 Queries

Class-5 queries have multiple source version and only one table. For this query class, we need to rewrite the input query Q into Q_i for each source version V_i and return the union of them as a rewritten query. If Q has n source versions, rather than repeating Class-1 reformulation n times, we extend the reformulation algorithm such that we perform a single reformulation producing multiple rewritten queries.

4.3 Class-6 Queries

For Class-6 queries, which have multiple source versions and multiple tables only with temporal joins, we can use the Class-5 reformulation to rewrite them. The results obtained by this is correct rewriting, since we know that temporal joins cannot be satisfied across the schema versions and we do not need to evaluate inter-schema-version temporal-joins: only intra-schema-version can satisfy temporal-joins. We therefore evaluate the input query on each source version and union them, which is essentially Class-5 reformulation.

4.4 Class-7 Queries

Class-7 queries are multiple source version and multiple table queries with non-temporal joins only. Because non-temporal joins may find its answers from inter-schema-version table joins, we must join the whole history of a table with the whole history of another table, which is more expensive than a union of intra-schema-version joins. The reformulatino algorithm is sketched as the following: for each accessed table T_i in Q, we create a subquery Q_{T_i}, that retrieves all attributes from the table T_i. Then, we rewrite Q_{T_i} into Q_{T_i}' using the Class-5 reformulation, and then return the join of all Q_{T_i}'.

4.5 Class-8 Queries

Class-8 queries are mixture of both Class-6 and Class-7 as these queries contain both temporal join and non-temporal join. Executing Class-7 reformulation will still find the correct rewriting, but not the most efficient one. To obtain a more efficient query, we exploit temporal joins in the query and use a hybrid of Class-6 and Class-7 reformulation, as follows:

Given a Class-8 query Q, we group the tables that are temporally joined and label them as G_i. We then make each G_i into a subquery Q_{G_i} and apply Class-6 reformulation to Q_{G_i}, obtaining Q_{G_i}'. For each table in Q that is not involved with any temporal join, we call it T_i and, for each T_i, we create Q_{T_i}' using Class-7 reformulation. Then, we finally return the (non-temporal) join of all rewritten subqueries, Q_{G_i}' and Q_{T_i}'.

5. OPTIMIZATION

We now present two optimization techniques that provide the efficiency and scalability of $\mathcal{PRIMA}$ reformulation, given the hundreds of schema versions to work with.

5.1 SMO Pruning

In the previous section, it has been shown that SMOs are translated into a set of XICs, which MARS uses to chase the input query. Even though MARS is a highly optimized chase engine, it does not scale with a very large number of schema versions because chase is inherently a very expensive computational task. Based on the observation that an input query, in general, does not access all tables or columns in schema history, we prune out SMOs that do not affect the tables or columns that the query accesses. This simple analysis allows us to prune many SMOs in general, which leads to a strong reduction in the number of XICs to be chased.

5.2 SMO Compression

SMOs have been designed with a philosophy that each SMO performs a minimal task so that users specify schema changes in a concise and modular way. In order to further reduce the chase task size, however, we compress several SMOs into a more expressive one, but still produces a single XIC. For example, RENAME COLUMN of three columns in a table will generate three XICs, if managed independently, several constraints to be chased by MARS. We compress them into one RENAME COLUMN MULTI, an SMO for internal-use only. Likewise, we compress multiple ADD COLUMN's and DROP COLUMN's into ADD COLUMN MULTI and DROP COLUMN MULTI, respectively.

6. EXPERIMENTAL STUDY

In this section, we evaluate the effectiveness of $\mathcal{PRIMA}$ approach, in terms of usability improvement, quality of rewritten queries, and rewriting cost. For experiment, we use synthetic data (the employee database given in Table 1) and also the real-world data from MediaWiki schema evolution history[8]. All experiments were performed on a Linux machine (Ubuntu 6.06) with two QuadCore Xeon 1.6GHz processors and 4GB memory. We used Sun Java 1.6.0 for $\mathcal{PRIMA}$ implementation and MonetDB/XQuery v0.22 for query execution.

6.1 Employee Database Schema Evolution

We have generated a data set for the employee database schema evolution given in Section 1.1. During the period between 2001-01-01 and 2005-12-31, we have five schema versions, each of which remains valid for one year (Jan. 1st

[8]All data sets and queries used in the experiment are available at `http://yellowstone.cs.ucla.edu/schema-evolution/index.php/Prima`

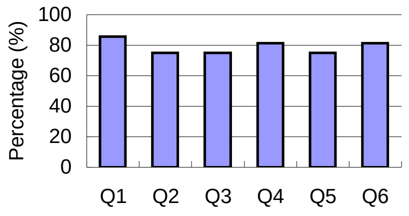


Figure 7: User Effort Saved for Query Writing

through Dec. 31st of each year). During each year, we update the data twice, and at each update point, we i) delete 1% of all employees, ii) update salary (for 25% of all employees), title (20%), and department (10%) and managers of chosen departments (50% of all departments), and iii) insert 10% of all employees newly. Over the history, we have records for 1,000 employees, 10 departments, and 4 titles in history, which makes 642 KB of MV-document.

In order to study the effectiveness of our taxonomy approach, we have designed twelve temporal queries: two queries for each of Class-1 (Q1, Q2), Class-2 (Q3, Q4), Class-3 (Q5, Q6), Class-5 (Q7, Q8), Class-6 (Q9, Q10), and Class-7 (Q11, Q12).

Using each of the twelve queries, we first verified the rewriting correctness by ensuring that i) results of the rewritten queries executed on the MV-document, and ii) results of the original queries executed on the migrated data into V_5, are equal.

We then measure the usability improvement by $\mathcal{PRIMA}$. Without $\mathcal{PRIMA}$, users need to find and access all historical schema versions, which contain the historical data that they want to query. Using $\mathcal{PRIMA}$, however, the process is much simplified as they can write queries using only one schema version of their choice and the system takes care of query rewriting. In order to objectively measure how much user effort each approach takes, we count each of the following as a unit cost: i) one (relational schema) table used in the query, ii) one (relational schema) column used in the query, iii) one SMOs affecting the tables and columns in the query, and iv) one period of schema versions[9]. Note that this is a similar metric to the one that was used in [28]. Figure 7 shows how $\mathcal{PRIMA}$ saves 75% or more of user effort in all six queries. The next six queries have the exactly same usability improvements and are omitted in the figure.

We now evaluate the benefit of our taxonomy-based approach, by measuring the performance of rewritten queries. We compare the execution times of rewritten queries produced by the following four methods.

- Baseline: Queries are reformulated without any query analysis results. Since it does not assume any information about the input queries, queries are rewritten using either i) Class-5 reformulation, if the queries have only one table, or ii) Class-7 reformulation, in case there are two or more tables accessed.

- MSF: Queries are rewritten with the query analysis results of MinSourceFind algorithm. An input query is reformulated using the Class-1 or Class-3 reformulation methods, in case the query has only one source version. If it has two or more source versions, the reformulation will be done for the minimal source versions computed by MinSourceFind.

- TJF: Queries are rewritten with the query analysis results of T-JoinFind algorithm. If a query has multiple source versions and also two or more tables, the query is further examined to see if it has temporal-joins only. If so, we may rewrite it using Class-6 reformulation.

- Both: Queries are rewritten with the query analysis results of both MinSourceFind and T-JoinFind algorithms. When both MSF and TJF are used, we fully characterize input queries into one of the eight query classes and the detailed query characteristics are exploited toward producing best rewritten queries possible.

[9]Users need to know the period of each schema version to decide whether they need to access that version.

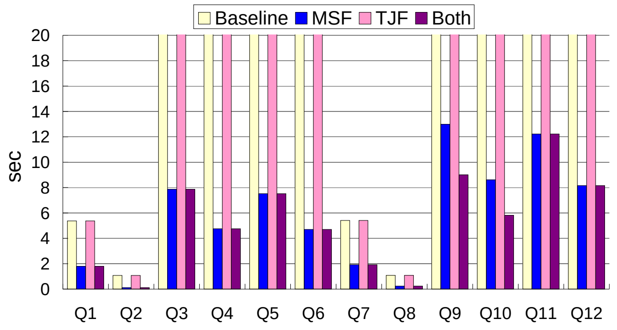


Figure 8: Execution Time of Rewritten Queries (All bars touching the highest grid represent out-of-memory error)

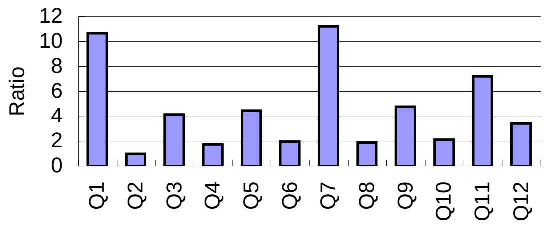


Figure 9: Ratio of Rewritten Query Performance to Original Query Performance on the Migrated Data

The result is shown in Figure 8. In most queries, baseline fails to finish, as it tries to answer the query in all five version, with expensive inter-schema-joins, which is prohibitively expensive for an XML DB that uses no index. MSF avoids this by detecting the minimal source versions and the produced rewritten queries finish in around 10 seconds in all cases. Q9 and Q10 shows the effectiveness of TJF, where the rewritten query performance is further improved in Both, compared to MSF. In these queries, MSF can tell that the minimal source versions are V_3, V_4, and V_5, but it can not guarantee whether the tables are temporally joined, so it has to perform Class-7 reformulation. TJF, however additionally detects that Q9 and Q10 have only temporal joins only and produce Class-6 reformulation.

We also examine the lower bound for the rewritten query performance as follows. As explained above, we perform data migration from all versions into the target version V_5, and execute the original query on it. Rewritten queries are expected to be slower, as they have to perform extra computation on the source versions, such as join and union introduced by COPY COLUMN and MERGE TABLE, respectively. On average, rewritten queries run 4.5 times slower than the migrated-data-queries, which is reasonable considering that the employee database schema evolution was designed to contain many COPY COLUMN in a small schema for illustration purpose. We observe that most of the extra processing is used toward computing the join of source version tables into the equivalent tables on the target version.

6.2 Wikipedia Schema Evolution

To demonstrate the practical completeness and scalability of our approach, we also test it against the challenging schema evolution history of MediaWiki[10]. MediaWiki is the open-source wiki software, originally developed to support the Wikipedia website[11], one of the ten most popular websites on the World Wide Web[12] and lately adopted by over 29,000 wiki-based websites (for a grand total of over 100 million pages)[13]. The MediaWiki web-portal software is developed in PHP and exploits a relational DB backend (MySQL by default) to store the website content and metadata. The underlying relational DB schema, during 4.5 years of development, has seen 171 schema revisions[14] and provides a rich and challenging dataset to test our system. The goal of this testing scenario is twofold: proving the practical completeness of the set of SMOs we defined w.r.t. real-life schema evolution and measure the rewriting time of our system against a long schema evolution history.

The MediaWiki schema evolution has been expressed in terms of SMOs, as shown in [9]. For queries, we derived real-world query workload to the Wikipedia site from the Wikipedia on-line profiler[15] and use the 20 most common queries in the Wikipedia. Since they are in SQL, we translated them into the temporal XQuery format. The $\mathcal{PRIMA}$ system were run to reformulate those queries, posed against the most recent schema version, into equivalent ones insisting on every schema version.

Figure 10 shows the rewriting time, as a function of the distance, in number of versions, between the target schema and the source schema. With reference to the optimization discussed in Section 5 we show the rewriting time for the unoptimized version of the system (baseline) and for two levels of optimization (SMO prune and SMO prune + SMO compress). In this scenario, due to the size of the schema (34 tables and 242 attributes in the last version), and the significantly high number of SMOs, the optimizations prove to be extremely effective, to such extent that the results are presented in logarithmic scale. The experiments conducted with the unoptimized $\mathcal{PRIMA}$ have been limited to a small number of cases (sampling and limiting the number of evolution steps) due to the extremely large execution time.

Figure 11 explains the improvement of reformulation time in Figure 10, where we show the number of SMOs used for query reformulation. While the number of SMOs considered by the unoptimized version of $\mathcal{PRIMA}$ grows rapidly with the distance between target and source schema, the number of SMOs used by the optimized versions is greatly reduced, thus delivering a significant performance improvement.

[10]http://www.mediawiki.org
[11]http://en.wikipedia.org
[12]Source: http://www.alexa.com.
[13]Source: http://s23.org/wikistats/.
[14]http://svn.wikimedia.org/viewvc/mediawiki/trunk/phase3/maintenance/tables.sql.
[15]http://noc.wikimedia.org/cgi-bin/report.py

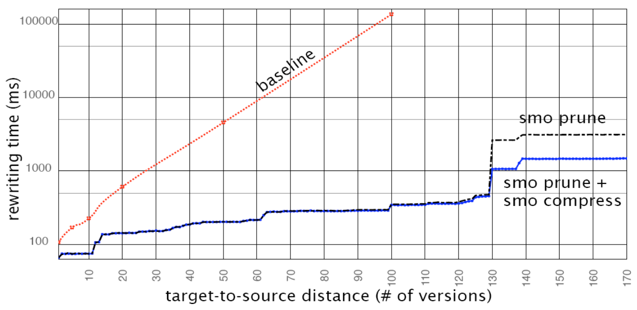


Figure 10: Query Reformulation Time in Wikipedia

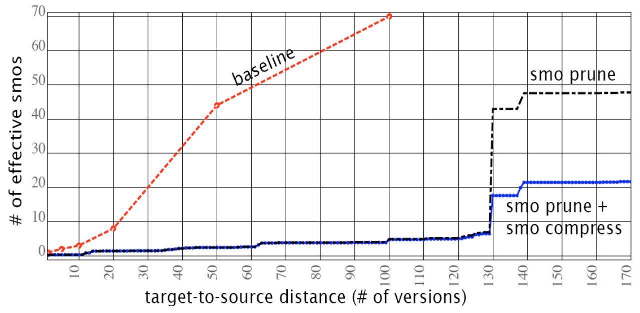


Figure 11: Number of SMOs Used for Query Reformulation in Wikipedia

7. RELATED WORK

Due to the practical importance of the problem, schema evolution has seen extensive research effort in the areas of (i) temporal database management and (ii) model management. During the late 80s and 90s, temporal database researchers have sought to support schema evolution. Their approaches are summarized as follows: i) preserving schema versions by means of timestamps, ii) letting users specify the schema version for query writing using SQL extension, and iii) supporting the query by migrating data to the queried version [16, 4, 7, 25]. Comprehensive survey appears in [22, 20]. These are valid solutions, as long as the data that need to be migrated remain relatively small in size. We provide a more practical solution for the problem by means of query reformulation, instead of data migration, into the source versions that are minimized by careful analysis of input queries.

More recently, schema evolution has also been studied in the framework of model management [5, 17], where model refers to metadata of various types, including relational and XML schemas, SQL view definitions, and mediator specifications. In [17], Melnik et al. present a prototype system called Rondo for a generic model management. The authors show that, using Rondo, model management is often feasible by using a small body of high-level code; however, the problem of evolving schema versions is not discussed in [17].

Lastly, we mention Panta Rhei framework [11] that seeks to provide an integrated support for schema evolution. In this framework, PRISM [10] supports the DBAs in the schema evolution process by managing and preserving user-provided SMOs to automate the tasks of data migration, legacy application query rewriting, and schema history recording.

8. CONCLUSIONS AND FUTURE WORK

Schema versioning for transaction-time databases had long been viewed as a solution to the schema-evolution problem that, although ideal in theory, in practice could not be effectively realized for real-life databases [22]. Our $\mathcal{PRIMA}$ prototype is changing the situation dramatically by building on two recent advances on database technology, whereby:

- XML and XQuery are i) well-supported in DBMS and ii) very conducive to temporal information management [13, 18, 21, 27].
- Powerful mapping techniques are now emerging [12, 26, 29, 30, 6] which have made it possible to map a query expressed against one schema into equivalent ones expressed on other schemas.

By exploiting these recent advances of database technology $\mathcal{PRIMA}$ aims at combining two design objectives that are not easily reconciled: one is the archivists' objective of achieving a faithful preservation by preserving the schema under which the records were first created; the second is the ease-of-use objective whereby users and applications can query the history of the database through the current schema, or any other schema version.

Currently, performance and scalability of $\mathcal{PRIMA}$ is limited by those of XQuery engine, and it can be improved by employing RDBMS-backed storage and query execution, as in [27]. We plan to explore this direction, to build a highly efficient and scalable transaction-time DBMS that provides a native support for schema evolution.

Acknowledgement

We thank Letizia Tanca and Vassilis Tsotras for many inspiring discussions, and Alex Chang, Uichin Lee, Barzan Mozafari, Ka Cheung Sia, and Hetal Thakkar for their comments on this and a previous version of this paper. We also thank the anonymous reviewers for their careful reading of the manuscript and useful comments and suggestions that resulted in an improved paper. This work was supported in part by NSF-IIS award 0705345: "III-COR: Collaborative Research: Graceful Evolution and Historical Queries in Information Systems – A Unified Approach".

9. REFERENCES

[1] Oracle Documentation. http://otn.oracle.com.
[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley Publishing Company, 1995.
[3] R. B. Almeida, B. Mozafari, and J. Cho. On the evolution of wikipedia. In *ICWSM*, 2007.
[4] G. Ariav. Temporally oriented data definitions - managing schema evolution in temporally oriented databases. *DKE*, 6(1):451–467, 1991.
[5] P. A. Bernstein. Applying model management to classical meta data problems. In *CIDR*, 2003.
[6] P. A. Bernstein, T. J. Green, S. Melnik, and A. Nash. Implementing mapping composition. In *VLDB*, 2006.
[7] C. D. Castro, F. Grandi, and M. R. Scalas. Schema versioning for multitemporal relational databases. *Information Systems*, 22(5):249–290, 1997.
[8] J. Clifford, A. Croker, F. Grandi, and A. Tuzhilin. On temporal grouping. In *Recent Advances in Temporal Databases*, pages 194–213. Springer Verlag, 1995.
[9] C. Curino, H. J. Moon, L. Tanca, and C. Zaniolo. Schema evolution in wikipedia: toward a web information system benchmark. In *ICEIS*, 2008.
[10] C. Curino, H. J. Moon, and C. Zaniolo. Graceful database schema evolution: the prism workbench. In *VLDB*, 2008.
[11] C. Curino, H. J. Moon, and C. Zaniolo. Managing the history of metadata in support for db archiving and schema evolution. In *ECDM*, 2008.
[12] A. Deutsch and V. Tannen. Mars: A system for publishing XML from mixed and redundant storage. In *VLDB*, 2003.
[13] D. Gao and R. T. Snodgrass. Temporal slicing in the evaluation of xml queries. In *VLDB*, 2003.
[14] S. Kepser. A proof of the turing-completeness of xslt and xquery. In *Technical report SFB 441, Eberhard Karls Universitat Tubingen*, 2002.
[15] S. Marche. Measuring the stability of data models. *European Journal of Information Systems*, 2(1):37–47, 1993.
[16] L. E. McKenzie and R. T. Snodgrass. Schema evolution and the relational algebra. *Information Systems*, 15(2):207–232, 1990.
[17] S. Melnik, E. Rahm, and P. A. Bernstein. Rondo: A programming platform for generic model management. In *SIGMOD*, 2003.
[18] A. O. Mendelzon, F. Rizzolo, and A. Vaisman. Indexing temporal XML documents. In *VLDB*, 2004.
[19] G. Özsoyoğlu and R. Snodgrass. Temporal and real-time databases: A survey. *tkde*, 7(4):513–532, 1995.
[20] S. Ram and G. Shankaranarayanan. Research issues in database schema evolution: the road not taken. In *Boston University School of Management, Department of Information Systems, Working Paper*, 2003.
[21] F. Rizzolo and A. Vaisman. Temporal xml: Modeling, indexing and query processing. *To appear in VLDB Journal.*
[22] J. Roddick. A Survey of Schema Versioning Issues for Database Systems. *Information and Software Technology*, 37(7):383–393, 1995.
[23] B. Shneiderman and G. Thomas. An architecture for automatic relational database system conversion. *ACM TODS*, 7(2):235–257, 1982.
[24] D. I. Sjoberg. Quantifying schema evolution. *Information and Software Technology*, 35(1):35–44, 1993.
[25] R. T. Snodgrass. *The TSQL2 Temporal Query Language.* Kluwer, 1995.
[26] Y. Velegrakis, R. J. Miller, and L. Popa. Mapping adaptation under evolving schemas. In *VLDB*, 2003.
[27] F. Wang, C. Zaniolo, and X. Zhou. Archis: An xml-based approach to transaction-time temporal database systems. *To appear in VLDB Journal.*
[28] C. Yu and H. V. Jagadish. Querying complex structured databases. In *VLDB*, 2007.
[29] C. Yu and L. Popa. Constraint-based XML query rewriting for data integration. In *SIGMOD*, 2004.
[30] C. Yu and L. Popa. Semantic adaptation of schema mappings when schemas evolve. In *VLDB*, 2005.

APPENDIX

A. TRANSLATION OF SMOS INTO XICS

In the following, we present the translation of user-provided SMOs into XICs, the schema mapping between two schema versions (See Section 4.1.2).

1. CREATE TABLE **R**: In forward direction, there is no source table, no mapping is necessary. Target table is created as empty. For backward direction, the XIC is as follows. $false$ indicates that the query accessing **R** in a new schema version cannot be satisfied in the old schema version, where the table did not exist.

   (CT_b) $[/v2db/S](y) \rightarrow false$

2. DROP TABLE **R**: After the table **R** is dropped, no query can access **R**, so no rewriting is needed for DROP TABLE and no XIC is generated for this SMO.

3. RENAME TABLE **R** INTO **S**: In forward direction (RT_f), we map a table-level element **R** in the old schema version to **S** in the new schema version such that they have identical table-level timestamps (i.e. s and e) and identical rows (i.e. x_2). To obtain a backward direction XIC (RT_b), we simply flip the body and the head of the forward XIC, which implies that no other source contribute to **S** in the new schema version. Note that the forward and the backward XICs together establish two-way inclusion dependencies, establishing an equivalence between two tables.

(RT_f)
$[/v1db/R](x_1), [./@ts](x_1, s), [./@te](x_1, e), [./row](x_1, x_2)$
$\rightarrow \exists y_1[/v2db/S](y_1), [./@ts](y_1, s), [./@te](y_1, e), [./row](y_1, x_2)$

(RT_b)
$[/v2db/S](y_1), [./@ts](y_1, s), [./@te](y_1, e), [./row](y_1, y_2)$
$\rightarrow \exists x_1[/v1db/R](x_1), [./@ts](x_1, s), [./@te](x_1, e), [./row](x_1, y_2)$

4. COPY TABLE **R** INTO **S**: In forward direction, the table **R** is mapped to the table **R** and table **S** in the new schema version. In backward direction, both **R** and **S** are mapped to **R** in the old schema version. Hence, it generates the following two XICs, in addition to (RT_f) and (RT_b) above.

   (OT_f) $[/v1db/R](x) \rightarrow [/v2db/R](x)$
   (OT_b) $[/v2db/R](y) \rightarrow [/v1db/R](y)$

5. DISTRIBUTE TABLE **R** INTO **S** WITH **cond2**, INTO **T** WITH **cond3**: DT_{f1} and DT_{b1} are the XICs for the table **S** in the new schema version, and DT_{f1} and DT_{b1} for the table **T**.

(DT_{f1})
$[/v1db/R](x_1), [./@ts](x_1, s), [./@te](x_1, e), [./row](x_1, x_2), cond2$
$\rightarrow \exists y_1[/v2db/S](y_1), [./@ts](y_1, s), [./@te](y_1, e), [./row](y_1, x_2)$

(DT_{f2})
$[/v1db/R](x_1), [./@ts](x_1, s), [./@te](x_1, e), [./row](x_1, x_2), cond3$
$\rightarrow \exists y_1[/v2db/T](y_1), [./@ts](y_1, s), [./@te](y_1, e), [./row](y_1, x_2)$

(DT_{b1})
$[/v2db/S](y_1), [./@ts](y_1, s), [./@te](y_1, e), [./row](y_1, y_2)$
$\rightarrow \exists x_1[/v1db/R](x_1), [./@ts](x_1, s), [./@te](x_1, e), [./row](x_1, y_2), cond2$

(DT_{b2})
$[/v2db/T](y_1), [./@ts](y_1, s), [./@te](y_1, e), [./row](y_1, y_2)$
$\rightarrow \exists x_1[/v1db/R](x_1), [./@ts](x_1, s), [./@te](x_1, e), [./row](x_1, y_2), cond3$

6. MERGE TABLE **S**, **T** INTO **R**: In forward direction, two tables are merged into one table. For the backward direction XIC, we flip the two forward XICs and obtain two XICs with the same head. We then merge the two bodies into a disjunctive, obtaining MT_b below.

(MT_{f1})
$[/v1db/S](x_1), [./@ts](x_1, s), [./@te](x_1, e), [./row](x_1, x_2)$
$\rightarrow \exists y_1[/v2db/R](y_1), [./@ts](y_1, s), [./@te](y_1, e), [./row](y_1, x_2)$

(MT_{f2})
$[/v1db/T](x_1), [./@ts](x_1, s), [./@te](x_1, e), [./row](x_1, x_2)$
$\rightarrow \exists y_1[/v2db/R](y_1), [./@ts](y_1, s), [./@te](y_1, e), [./row](y_1, x_2)$

(MT_b)
$[/v2db/R](y_1), [./@ts](y_1, s), [./@te](y_1, e), [./row](y_1, y_2)$
$\rightarrow \exists x_1[/v1db/S](x_1), [./@ts](x_1, s), [./@te](x_1, e), [./row](x_1, y_2) \lor$
$[/v1db/T](x_1), [./@ts](x_1, s), [./@te](x_1, e), [./row](x_1, y_2)$

7. ADD COLUMN **A** AS **const** INTO **R**: Assuming that **R** had columns $\mathbf{C}_1$, $\mathbf{C}_2$, ..., and $\mathbf{C}_N$ before the schema change: Also assume that T_s and T_e are start time and end time of the source schema version.

(AC_f)
$[/v1db/R](x_1), [./@ts](x_1, s_1), [./@te](x_1, e_1),$
$[./row](x_1, x_2), [./@ts](x_2, s_2), [./@te](x_2, e_2),$
$[./C_1](x_2, z_1), [./C_2](x_2, z_2), ..., [./C_N](x_2, z_N)$
$\rightarrow \exists y_1, y_2, y_3[/v2db/S](y_1), [./@ts](y_1, s_1), [./@te](y_1, e_1),$
$[./row](y_1, y_2), [./@ts](y_2, s_2), [./@te](y_2, e_2),$
$[./A](y_2, y_3), [./@ts](y_3, T_s), [./@te](y_3, T_e), [./text()](y3, const),$
$[./C_1](y_2, z_1), [./C_2](y_2, z_2), ..., [./C_N](y_2, z_N)$

(AC_b)
$[/v2db/S](y_1), [./@ts](y_1, s_1), [./@te](y_1, e_1),$
$[./row](y_1, y_2), [./@ts](y_2, s_2), [./@te](y_2, e_2),$
$[./A](y_2, y_3), [./@ts](y_3, T_s), [./@te](y_3, T_e), [./text()](y3, const),$
$[./C_1](y_2, z_1), [./C_2](y_2, z_2), ..., [./C_N](y_2, z_N)$
$\rightarrow \exists x_1, x_2[/v1db/R](x_1), [./@ts](x_1, s_1), [./@te](x_1, e_1),$
$[./row](x_1, x_2), [./@ts](x_2, s_2), [./@te](x_2, e_2),$
$[./C_1](x_2, z_1), [./C_2](x_2, z_2), ..., [./C_N](x_2, z_N)$

8. DROP COLUMN **A** FROM **R**: Assuming that **R** had columns **A**, $\mathbf{C}_1$, $\mathbf{C}_2$, ..., and $\mathbf{C}_N$ before the schema change:

(DC_f)
$[/v1db/R](x_1), [./@ts](x_1, s_1), [./@te](x_1, e_1),$
$[./row](x_1, x_2), [./@ts](x_2, s_2), [./@te](x_2, e_2),$
$[./A](x_2, x_3), [./@ts](x_3, s_3), [./@te](x_3, e_3),$
$[./C_1](x_2, z_1), [./C_2](x_2, z_2), ..., [./C_N](x_2, z_N)$
$\rightarrow \exists y_1, y_2[/v2db/S](y_1), [./@ts](y_1, s_1), [./@te](y_1, e_1),$
$\quad [./row](y_1, y_2), [./@ts](y_2, s_2), [./@te](y_2, e_2),$
$\quad [./C_1](y_2, z_1), [./C_2](y_2, z_2), ..., [./C_N](y_2, z_N)$

(DC_b)
$[/v2db/S](y_1), [./@ts](y_1, s_1), [./@te](y_1, e_1),$
$[./row](y_1, y_2), [./@ts](y_2, s_2), [./@te](y_2, e_2),$
$[./C_1](y_2, z_1), [./C_2](y_2, z_2), ..., [./C_N](y_2, z_N)$
$\rightarrow \exists x_1, x_2[/v1db/R](x_1), [./@ts](x_1, s_1), [./@te](x_1, e_1),$
$\quad [./row](x_1, x_2), [./@ts](x_2, s_2), [./@te](x_2, e_2),$
$\quad [./A](x_2, x_3), [./@ts](x_3, s_3), [./@te](x_3, e_3),$
$\quad [./C_1](x_2, z_1), [./C_2](x_2, z_2), ..., [./C_N](x_2, z_N)$

9. RENAME COLUMN **A** IN **R** TO **B**: Assuming that **R** has columns **A**, $\mathbf{C}_1$, $\mathbf{C}_2$, ..., and $\mathbf{C}_N$

(RC_f)
$[/v1db/R](x_1), [./@ts](x_1, s_1), [./@te](x_1, e_1),$
$[./row](x_1, x_2), [./@ts](x_2, s_2), [./@te](x_2, e_2),$
$[./A](x_2, x_3), [./@ts](x_3, s_3), [./@te](x_3, e_3), [./text()](x_3, tx_3)$
$[./C_1](x_2, z_1), [./C_2](x_2, z_2), ..., [./C_N](x_2, z_N)$
$\rightarrow \exists y_1, y_2, y_3[/v2db/S](y_1), [./@ts](y_1, s_1), [./@te](y_1, e_1),$
$\quad [./row](y_1, y_2), [./@ts](y_2, s_2), [./@te](y_2, e_2),$
$\quad [./B](y_2, y_3), [./@ts](y_3, s_3), [./@te](y_3, e_3), [./text()](y_3, tx_3)$
$\quad [./C_1](y_2, z_1), [./C_2](y_2, z_2), ..., [./C_N](y_2, z_N)$

(RC_b)
$[/v2db/S](y_1), [./@ts](y_1, s_1), [./@te](y_1, e_1),$
$[./row](y_1, y_2), [./@ts](y_2, s_2), [./@te](y_2, e_2),$
$[./B](y_2, y_3), [./@ts](y_3, s_3), [./@te](y_3, e_3), [./text()](y_3, tx_3)$
$[./C_1](y_2, z_1), [./C_2](y_2, z_2), ..., [./C_N](y_2, z_N)$
$\rightarrow \exists x_1, x_2, x_3[/v1db/R](x_1), [./@ts](x_1, s_1), [./@te](x_1, e_1),$
$\quad [./row](x_1, x_2), [./@ts](x_2, s_2), [./@te](x_2, e_2),$
$\quad [./A](x_2, x_3), [./@ts](x_3, s_3), [./@te](x_3, e_3), [./text()](x_3, tx_3)$
$\quad [./C_1](x_2, z_1), [./C_2](x_2, z_2), ..., [./C_N](x_2, z_N)$

10. COPY COLUMN **C** FROM **R** INTO **S** WHERE **R.A** = **S.A**: The following two XICs, as well as the identity mapping for **R** as it is not modified by this SMO.

(CC_f)
$[/v1db/R](x_1), [./@ts](x_1, xs_1), [./@te](x_1, xe_1),$
$[./row](x_1, x_2), [./@ts](x_2, xs_2), [./@te](x_2, xe_2),$
$[./A](x_2, x_3), [./@ts](x_3, xs_3), [./@te](x_3, xe_3), [./text()](x_3, xt_3),$
$[./C](x_2, x_4), [./@ts](x_4, xs_4), [./@te](x_4, xe_4), [./text()](x_4, xt_4),$
$[/v1db/S](y_1), [./@ts](y_1, ys_1), [./@te](y_1, ye_1),$
$[./row](y_1, y_2), [./@ts](y_2, ys_2), [./@te](y_2, ye_2),$
$[./A](y_2, y_3), [./@ts](y_3, ys_3), [./@te](y_3, ye_3), [./text()](y_3, yt_3),$
$[./C_1](y_2, yc_1), [./C_2](y_2, yc_2), ..., [./C_N](y_2, yc_N),$
$xt_3 = yt_3$
$\rightarrow \exists z_1, z_2, z_3, z_4[/v2db/S](z_1), [./@ts](z_1, ys_1), [./@te](z_1, ye_1),$
$\quad [./row](z_1, z_2), [./@ts](z_2, ys_2), [./@te](z_2, ye_2),$
$\quad [./A](z_2, z_3), [./@ts](z_3, ys_3), [./@te](z_3, ye_3), [./text()](z_3, yt_3)$
$\quad [./C](z_2, z_4), [./@ts](z_4, xs_4), [./@te](z_4, xe_4), [./text()](z_4, xt_4)$
$\quad [./C_1](z_2, zc_1), [./C_2](z_2, zc_2), ..., [./C_N](z_2, zc_N)$

(CC_b)
$[/v2db/S](z_1), [./@ts](z_1, ys_1), [./@te](z_1, ye_1),$
$[./row](z_1, z_2), [./@ts](z_2, ys_2), [./@te](z_2, ye_2),$
$[./A](z_2, z_3), [./@ts](z_3, ys_3), [./@te](z_3, ye_3), [./text()](z_3, yt_3)$
$[./C](z_2, z_4), [./@ts](z_4, xs_4), [./@te](z_4, xe_4), [./text()](z_4, xt_4)$
$[./C_1](z_2, zc_1), [./C_2](z_2, zc_2), ..., [./C_N](z_2, zc_N)$
$\rightarrow \exists x_1, x_2, x_3, x_4, y_1, y_2, y_3, xs_1, xe_1, xs_2, xe_2, xs_3, xe_3, xt_3,$
$\quad\quad yc_1, yc_2, ..., yc_N$
$\quad [/v1db/R](x_1), [./@ts](x_1, xs_1), [./@te](x_1, xe_1),$
$\quad [./row](x_1, x_2), [./@ts](x_2, xs_2), [./@te](x_2, xe_2),$
$\quad [./A](x_2, x_3), [./@ts](x_3, xs_3), [./@te](x_3, xe_3), [./text()](x_3, xt_3),$
$\quad [./C](x_2, x_4), [./@ts](x_4, xs_4), [./@te](x_4, xe_4), [./text()](x_4, xt_4),$
$\quad [/v1db/S](y_1), [./@ts](y_1, ys_1), [./@te](y_1, ye_1),$
$\quad [./row](y_1, y_2), [./@ts](y_2, ys_2), [./@te](y_2, ye_2),$
$\quad [./A](y_2, y_3), [./@ts](y_3, ys_3), [./@te](y_3, ye_3), [./text()](y_3, yt_3),$
$\quad [./C_1](y_2, yc_1), [./C_2](y_2, yc_2), ..., [./C_N](y_2, yc_N),$
$\quad xt_3 = yt_3$

11. MOVE COLUMN **A** FROM **R** INTO **S** WHERE **R.A** = **S.A**: same as COPY COLUMN rules, without the identity rules.