A Relational Multi-Schema Data Model and Query Language for Full Support of Schema Versioning*

Fabio Grandi

CSITE-CNR and DEIS, Alma Mater Studiorum – Università di Bologna
Viale Risorgimento 2, 40136 Bologna, Italy, email: fgrandi@deis.unibo.it

Abstract. Schema versioning is a powerful tool not only to ensure reuse of data and continued support of legacy applications after schema changes, but also to add a new degree of freedom to database designers, application developers and final users. In fact, different schema versions actually allow one to represent, in full relief, different *points of view* over the modelled application reality. The key to such an improvement is the adoption of a *multi-pool* implementation solution, rather that the single-pool solution usually endorsed by other authors. In this paper, we show some of the application potentialities of the multi-pool approach in schema versioning through a concrete example, introduce a simple but comprehensive logical storage model for the mapping of a multi-schema database onto a standard relational database and use such a model to define and exemplify a multi-schema query language, called MSQL, which allows one to exploit the full potentialities of schema versioning under the multi-pool approach.

1 Introduction

However careful and accurate the initial design may have been, a database schema is likely to undergo changes and revisions after implementation. In order to avoid the loss of data after schema changes, **schema evolution** has been introduced to provide (partial) automatic recovery of the extant data by adapting them to the new schema. However, if only the updated schema is retained, all the applications compiled with the past schema may cease to work. In order to let applications work on multiple schemata (at least to attain continued support of legacy applications), schema evolution is not sufficient and the maintenance of more than one schema is required. This leads to the notion of **schema versioning** and **schema version** which is the persistent outcome of the application of schema modifications [6, 5].

Several issues and design options concerning schema versioning support in a relational database have been studied in [2]. In particular, two implementation options, named *single-pool* and *multi-pool* solutions, were considered for the management of the extensional data. We briefly recall here the difference between these options:

single-pool : all the schema versions are associated with a unique, shared, extensional repository, so that the same objects cannot have different values for the same properties when "viewed" through different schema versions;

multi-pool : each schema version is associated with a "private" extensional data pool; different data pools may contain the same objects having (possibly) completely independent representations and evolutions.

* This work has been partially supported by the Italian MURST Project "D2I".

Although the multi-pool solution may look more flexible even at a first glance, a single-pool solution has usually been considered satisfactory for implementation, for example by means of the so-called *completed schema* technique [7, 2], as it limits the storage space overhead due to coexistence of multiple schemas.

In our opinion, with a single-pool implementation solution, the schema versioning approach gives up, from a conceptual point of view, most of its potential appeal and usefulness. As a matter of fact, if the single-pool is implemented via the completed schema solution, the schema versioning support trivially reduces to a *view mechanism*, as schema versions can be defined as collections of simple projection-based views (the schema of a relation in each schema version is, by definition, a subset of the completed schema). Moreover, a database is an abstract representation (model) of a portion of real world of interest for some application (mini-world). Such a representation includes an *intensional* aspect –i.e. the schema– and an *extensional* aspect –i.e. the instance. A unique point of view on the mini-world is implied in this conceptualization. When we consider the useful coexistence of different schema versions for the same database, we actually introduce *different viewpoints* on the mini-world, which are, in some sense, all useful for the applications. To have different viewpoints means that the very same mini-world objects have different representations in the system: if this "multiple modeling" has an intensional aspect, it also has, in general, an extensional aspect: different viewpoints may render the same objects endowed with different structural properties but also with different values for the same properties. By the way, the notion of useful "versions" originated in the field of OODBs used for design/engineering applications [3, 8], where the main focus was on different versions of the same extant objects. However, for the sake of generality, such a difference was allowed to span also structural properties, giving rise to the introduction of schema versions. When we proposed the multi-pool solution in [1, 2], we also aimed at giving back the right perspective on versioning, stating that different representations at intensional level –i.e. schema versioning– may imply, in general, useful different representations also at extensional level.

On the other hand, the main objection that can be moved against the multi-pool solution is the potentially huge storage space overhead due to the introduction of multiple representations of each object in the database: more or less, the full database extent needs to be replicated in every pool. First, our more direct answer to this objection relies on the progress of the secondary memory technology, that has been making the multi-pool storage space requirements more and more inexpensive. For instance, with respect to the storage cost of a database when we introduced the multi-pool in [1], current technology could enable today the storage of the same database with more than **100** schema versions and data pools, at the same cost or less (as a 20GB disk costs today much less than a 200MB disk in 1995). This number may exceed far enough the number of schema versions probably required by any reasonable application. Second, not all the data actually need to be replicated in every data pool: for the unchanged attributes and attribute values of the objects which belong to several data pools, a non-redundant storage representation can be adopted, possibly in conjunction with lazy-style update techniques to delay space duplication till when it becomes strictly necessary. With such kinds of optimizations, storage space requirements may resemble those of the single-pool solution, and the overhead is introduced to strictly support the enhanced semantics of the multiple versions connected to different schema versions. Anyway, physical optimization is beyond the scope of this paper, and storage requirements will not be considered any longer here. We rather focus on

logical aspects, to highlight the multi-pool solution potentialities and show its feasibility with our data model and query language definitions.

To this purpose, we will introduce in Sec. 3 a data model, named *logical storage model*, which maps attributes and tables in a multi-pool schema versioning setting to traditional relational tables, on the basis of which the semantics of a Multi-schema Query Language (MSQL) will be defined in Sec. 4. The specification of a query language supporting *multi-schema* queries, that is queries where data from different schema versions and data pools can be freely combined, is the novel contribution of this paper. In previous works on schema versioning, query functionalities were limited to accessing data stored according to a schema version through a different schema version, which gives only a small portion of the whole picture. Undoubtedly, one of the reasons why the multi-pool solution could not be appreciated so far was the lack of a multi-schema query language. In order to show how some of the functionalities of schema versioning and of the multi-pool approach could be valuable in a concrete environment, an initial example is also given in the Section that follows.

2 A Concrete Example

Assume "ACME Motors" is a car manufacturer based in the United States and selling cars in several countries world-wide by means of a Web-based e-commerce platform. We consider the relational DB supporting the e-commerce activity, with an initial schema version **USMKT** designed for the American market. Let **CAR(NAME,PRICE)** be the initial schema of a relation storing the car catalogue in **USMKT**, which is populated with the following instance:

CAR

NAME	PRICE
Bomb 3.0	35K
Lark 2.0	20K
Lark 2.5	26K

where prices are expressed in US dollars. Now we consider a different schema version **EUMKT** designed for the European market, where the consumers may be interested in knowing the anti air pollution compliance rule of each car (rules are defined as "Euro1", "Euro2" and so on; the higher the number, the more restrictive the rule). To this purpose, a new column, say **APC** (for Air Pollution Compliance), is added to table **CAR** via the schema change:

```
alter table CAR add column APC
```

producing the relation schema **CAR(NAME,PRICE,APC)** in the new schema version **EUMKT**. The data pool associated with **EUMKT** is initially populated as the one associated with **USMKT**, with the new column padded by null values. However, "ACME Motors" has judged marketing the "Bomb 3.0" model in Europe non convenient and, thus, the corresponding tuple is removed from the data pool. Moreover, as far as car prices are concerned, they must be expressed in Euros rather than in US Dollars as in **USMKT**. However, their new values are defined not only according to the currency exchange rates, but rather on the basis of a different marketing strategy, targeted for Europe: such a strategy takes into account possibly different production and logistic costs but also actual different market conditions (e.g. usually, similar cars cost more in Europe than in the US). After also some extensional operations have been effected and the

results validated, we assume the resulting **CAR** table in the data pool associated with **EUMKT** is the following:

CAR

NAME	PRICE	APC
Lark 2.0	24K	Euro4
Lark 2.5	32K	Euro4

Notice from the example that, as a consequence of the independent modifications that can be made on extensional data, the data pools do not contain, in general, the same objects (in the example, the object "Bomb 3.0" has been deleted wrt **EUMKT**, others could be added) and the same objects are allowed to have different values for the same attributes. *The adoption of the multi-pool solution is actually a consequence of this application requirement.* As a matter of fact, the coexistence of the two **CAR** table instances shown in our example requires the existence of two distinct data pools associated with the schema versions **USMKT** and **EUMKT**, with an independent management and evolution of the data instances associated, in general, to the different schema versions. We emphasize the fact that, in our example, no common representation could be given for the column **PRICE** as required in the single-pool approach, since no automatic *conversion function* can be defined to uniformly compute the European price of a car model from the American one, and vice versa: in the example, a different percentage gap between the two engine equipments of the Lark model is applied. *Different schema versions and different data pools are needed since we have different points of view on the same data (marketing strategies are part of these points of view). New attribute values are set by explicit updates on data pool underlying the new schema version.*

We continue the example by supposing that "ACME Motors" wants also to provide localized versions of its information system and web site, for the convenience of its national subsidiaries (e.g. many Italian customers do not speak English and, for example, also several Italian software developers would prefer dealing with Italian names in the database schema). Therefore, relation and attribute names are translated into the corresponding national Language for non English speaking countries. The schema changes that follow are used to produce the Italian version **ITMKT**:

```
alter table CAR alter column NAME rename as NOME;
alter table CAR alter column PRICE rename as PREZZO;
alter table CAR alter column APC rename as NAI;
alter table CAR rename as AUTO;
```

We assume the table in the new schema version, as long as its instance in the corresponding data pool, eventually come out as:

AUTO

NOME	PREZZO	NAI
Lark 2.0	25K	Euro4
Lark 2.5	31K	Euro3

It can be noticed that prices have been slightly revised due to a targeted marketing policy for Italy. First of all, cars are sold at higher prices than in other EU countries (e.g. due to

different tax rates); this explains the increase in the "Lark 2.0" price with respect to the **EUMKT** version (still used in an English speaking country, say in Eire). Second, it was judged that the model "Lark 2.5" could be purchased in Italy also if does not comply with the more restrictive "Euro4" rule (a potential buyer could be more interested in high performance and an appealing price, for a car which is anyway high-powered with respect to the market average). Hence, the adoption of a less sophisticated anti-pollution device lead to a final cost restraint.

Finally, we may consider the fact that one car model could also be marketed with different names in different countries. For example, the "Lark 2.5" model could be sold in Italy as "Lark GT". To this end, the **NOME** value needs to be changed in the pool connected with **ITMKT**. Notice that this is a change of the *primary key* value. In order to maintain identity of data objects across different data pools whatever it happens to them, including key changes, we will use an abstract identification method based on *surrogates*. Surrogates are assumed to be system-managed identifiers, whose uniqueness in the whole database is guaranteed. If surrogates have already been used (e.g. in the TSQL2 design [4]) to allow different (extensional) versions of the same object to differ in the key value, we can use them to also allow schema changes involving the definition of the key [9, 7], which gives maximum flexibility to the schema maintenance process.

3 A Multi-schema Relational Data Model

We present in this Section a *logical storage model* for multi-pool schema versioning support in a relational system. In this model, a multi-pool database schema is a collection $DB = \{SV_0, SV_1, SV_2, \ldots, SV_n\}$ of schema versions, where SV_0 is the initial database schema and SV_n is the most recently created schema version. Each schema version $SV_i \in DB$ contains five "normal" tables, which provide a combined intensional-extensional representation of the corresponding data pool structure and contents. In fact, the data structures associated to the data pool SV_i are defined in our model by the relational schema:

$$\begin{array}{l}\mathsf{RName}_i(\underline{\mathsf{RID}}, \mathsf{Name}) \\ \mathsf{AName}_i(\underline{\mathsf{AID}}, \mathsf{Name})\end{array} \quad \mathsf{RSchema}_i(\mathsf{RID}, \underline{\mathsf{AID}}) \quad \begin{array}{l}\mathsf{RExt}_i(\mathsf{RID}, \underline{\mathsf{TID}}) \\ \mathsf{AExt}_i(\underline{\mathsf{TID}}, \underline{\mathsf{AID}}, \mathsf{Value})\end{array}$$

Attributes RID, AID and TID are *surrogates* that are used to uniquely identify a relation, an attribute and a single tuple, respectively. The RName_i and AName_i tables associate relations and attributes with their names in SV_i. $\mathsf{RSchema}_i$ (RExt_i) associate to every relation the attributes (tuples) in their schema (instance) in the SV_i data pool. AExt_i associates attribute values to the tuples of all the relations.

With reference to our example, the **CAR** relation in schema version **USMKT**, that we can rewrite to highlight a possible assignment of the required surrogates (shown with Greek letters) as follows :

CAR (ρ_1)

TID	**NAME** (α_1)	**PRICE** (α_2)
(τ_1)	**Bomb 3.0**	**35K**
(τ_2)	**Lark 2.0**	**20K**
(τ_3)	**Lark 2.5**	**26K**

can be represented in our logical storage model as:

$\mathsf{RName}_{\mathbf{USMKT}}$

RID	Name
ρ_1	**CAR**

$\mathsf{AName}_{\mathbf{USMKT}}$

AID	Name
α_1	**NAME**
α_2	**PRICE**

$\mathsf{RSchema}_{\mathbf{USMKT}}$

RID	AID
ρ_1	α_1
ρ_1	α_2

$\mathsf{RExt}_{\mathbf{USMKT}}$

RID	TID
ρ_1	τ_1
ρ_1	τ_2
ρ_1	τ_3

$\mathsf{AExt}_{\mathbf{USMKT}}$

TID	AID	Value
τ_1	α_1	**Bomb 3.0**
τ_1	α_2	**35K**
τ_2	α_1	**Lark 2.0**
τ_2	α_2	**20K**
τ_3	α_1	**Lark 2.5**
τ_3	α_2	**26K**

Notice that such relations contain mixed intensional and extensional information: whereas RName_i, AName_i and $\mathsf{RSchema}_i$ basically contain catalog information about the schema of each relation, RExt_i and AExt_i define the composition of the instance of every relation. The relation $\mathsf{RSchema}_i$ is necessary to maintain the information about the schema of a relation when its extension is empty (when it is not empty, the schema could also be derived from $\pi_{\mathsf{RID,AID}}(\mathsf{RExt}_i \bowtie \mathsf{AExt}_i)$).

We called our model a *logical storage model* as its definitions can be used, as a plain relational schema, to implement the schema version SV_i and the data pool connected with SV_i on top of a conventional relational database. Such an implementation, though not optimized, is straightforward and, thus, can easily be used for fast prototyping of our complete model on top of a state-of-the-art DBMS. In order to optimize the storage (and make each of our data pools more similar to a "normal" database), the join result $\mathsf{RExt}_i \bowtie \mathsf{AExt}_i$ could be horizontally partitioned according to the different values of RID. In such a way, each of the resulting fragments would contain the same data as in the instance of a single relation (instead of having all of them mixed and distributed in RExt_i and AExt_i). Anyway, the problem of a physically optimized implementation of our model is out of the scope of the present research.

Notice that the "real" relation **CAR** cannot be defined as a plain SQL view over the logical storage model tables, as it would require to construct SQL queries using the results of other queries in their syntax and some manipulation operation which is not expressible with SQL.

4 A Multi-schema Query Language

We introduce here the principal syntactic peculiarities and the semantics of the Multi-schema Query Language MSQL, and give some examples of its use. MSQL allows users to express *multi-schema* queries, which involve intensional (resp. extensional) data belonging to different schema versions (resp. data pools) at the same time.

The basic "tool" in the MSQL syntax that extends SQL is a way to contextualize names and data references to schema versions. In particular, we adopt the syntax "[SV:X]" to denote the conceptual entity (relation or attribute) that was named "X" in schema version "SV", and the syntax "SV:X" to denote the extension with respect to schema version "SV" of the conceptual entity "X". On the other hand, the expression "the extension of X wrt schema version SV" means the value of X in the data pool associated to SV. In the complete expression "SV_i:[SV_j:X]", we call "SV_i:" and "SV_j:" an *extensional qualifier* and a *naming qualifier*, respectively.

For instance, the intended meaning of the MSQL queries which follow:

```
select * from [SV_j:R]                                   (Q_1)
select * from SV_i:R                                     (Q_2)
select * from SV_i:[SV_j:R]                              (Q_3)
```

is:

- Q_1: retrieve (the current contents of) the table that was called R in schema version SV_j;
- Q_2: retrieve the contents wrt schema version SV_i of the table (currently called) R;
- Q_3: retrieve the contents wrt schema version SV_i of the table that was called R in schema version SV_j.

Here and after, "current" has the actual meaning of "wrt to the *currently selected* schema version". Similarly, the meaning of the MSQL queries:

```
select [SV_j:A] from R                                   (Q_4)
select SV_i:A from R                                     (Q_5)
select SV_i:[SV_j:A] from R                              (Q_6)
```

is the following:

- Q_4: retrieve (the current contents of) the column that was called A in schema version SV_j of the table (currently called) R;
- Q_5: retrieve the contents wrt schema version SV_i of the column (currently called) A of the table (currently called) R;
- Q_6: retrieve the contents wrt schema version SV_i of the column that was called A in schema version SV_j of the table (currently called) R.

These forms can also be combined, and the most general form in which a column of a relation can be involved in a query (together with four different schema versions) is the following:

```
select SV_i:[SV_j:A] from SV_k:[SV_ℓ:R]                  (Q_7)
```

whose intended semantics is:

- Q_7: retrieve the values wrt schema version SV_i of the attribute that was called A in schema version SV_j, in the tuples that belong wrt schema version SV_k to the extension of the table that was called R in schema version SV_ℓ.

If an explicit extensional qualifier before attribute names (like SV_i in Q_7) is omitted (in the **`select`** and **`where`** clauses), it is implicitly assumed to be the same extension qualifier used in the **`from`** clause for the relation the attribute belongs to (e.g. SV_k in Q_7). The other naming and extensional qualifiers have "wrt to the *currently selected* schema version" as default. For instance, the meaning of the sample query:

$$\texttt{select [SV}_1\texttt{:A] from SV}_2\texttt{:R} \qquad (Q_8)$$

is:

– Q_8: retrieve the contents wrt schema version SV_2 of the column that was called A in schema version SV_1 of the table (currently called) R.

We introduce now the semantics of a general MSQL Select-Project-Join (SPJ) query. For the sake of simplicity, we start from the simplified case where extensional qualifiers in attribute expressions are omitted. Due to the default rule stated above, in this case relations and attributes are identified by means of the naming qualifiers, and then the query is executed on the relation versions belonging to the schema versions identified by the extensional qualifiers in the **`from`** clause. Hence, the required attributes values for each relation are extracted from the tuples belonging to such relation extensions. The general form of this kind of MSQL queries can be expressed as:

$$\begin{array}{l}\texttt{select S}_{p_1}\texttt{.[SV}_{j_1}\texttt{:A}_1\texttt{], ..., S}_{p_m}\texttt{.[SV}_{j_m}\texttt{:A}_m\texttt{]} \\ \quad\texttt{from SV}_{k_1}\texttt{:[SV}_{\ell_1}\texttt{:R}_1\texttt{] AS S}_1\texttt{, ..., SV}_{k_n}\texttt{:[SV}_{\ell_n}\texttt{:R}_n\texttt{] AS S}_n \\ \quad\texttt{where } F(\texttt{S}_{p'_1}\texttt{.[SV}_{j'_1}\texttt{:A}_{q_1}\texttt{]},\ldots,\texttt{S}_{p'_r}\texttt{.[SV}_{j'_r}\texttt{:A}_{q_r}\texttt{]})\end{array} \qquad (Q_9)$$

Notice that, in order to avoid ambiguity, aliases $S_1, \ldots, S_n$ have been introduced for the relation expressions in the **`from`** clause, and $\{p_1, \ldots, p_m\} \cup \{p'_1, \ldots, p'_r\} \subseteq \{1, \ldots, n\}$ (i.e. the relation aliases appearing in attribute expressions are among those declared in the **`from`** clause). Moreover, a standard SQL dotted notation $S_i.X$ has been used (in the **`select`** and **`where`** clauses) to denote which relation each attribute belongs to. The expression F in the **`where`** clause is assumed, as usual, to be a Boolean expression of attribute comparisons (local and join predicates). The precise semantics of the MSQL query Q_9 can be given in tuple calculus (first-order logic) as follows:

$$\begin{array}{lll}\{\,(v_1,\ldots,v_m) \mid & \exists r_1 \cdots \exists r_n\, \exists s_1 \cdots \exists s_n\, \exists a_1 \cdots \exists a_m\, \exists a_{q_1} \cdots \exists a_{q_r}\, \exists v_{m+1} \cdots \exists v_{m+r} & (1)\\ & \mathsf{RName}_{\ell_1}(r_1, R_1) \wedge \ldots \wedge \mathsf{RName}_{\ell_n}(r_n, R_n) \wedge & (2)\\ & \mathsf{RExt}_{k_1}(r_1, s_1) \wedge \ldots \wedge \mathsf{RExt}_{k_n}(r_n, s_n) \wedge & (3)\\ & \mathsf{AName}_{j_1}(a_1, A_1) \wedge \ldots \wedge \mathsf{AName}_{j_m}(a_m, A_m) \wedge & (4)\\ & \mathsf{RSchema}_{j_1}(r_{p_1}, a_1) \wedge \ldots \wedge \mathsf{RSchema}_{j_m}(r_{p_m}, a_m) \wedge & (5)\\ & \mathsf{AExt}_{k_{p_1}}(s_{p_1}, a_1, v_1) \wedge \ldots \wedge \mathsf{AExt}_{k_{p_m}}(s_{p_m}, a_m, v_m) \wedge & (6)\\ & \mathsf{AName}_{j'_1}(a_{q_1}, A_{q_1}) \wedge \ldots \wedge \mathsf{AName}_{j'_r}(a_{q_r}, A_{q_r}) \wedge & (7)\\ & \mathsf{RSchema}_{j'_1}(r_{p'_1}, a_{q_1}) \wedge \ldots \wedge \mathsf{RSchema}_{j'_r}(r_{p'_r}, a_{q_r}) \wedge & (8)\\ & \mathsf{AExt}_{k_{p'_1}}(s_{p'_1}, a_{q_1}, v_{m+1}) \wedge \ldots \wedge \mathsf{AExt}_{k_{p'_r}}(s_{p'_r}, a_{q_r}, v_{m+r}) \wedge & (9)\\ & F(v_{m+1}, \ldots, v_{m+r})\,\} & (10)\end{array}$$

As to the variables in (1), r_i, a_j, s_k, v_ℓ denote, respectively, a relation identifier, an attribute identifier, a tuple identifier and an atomic value. Clauses in (2) bind relation identifiers (e.g. r_i is the relation named R_i in SV_{ℓ_i}). Clauses in (3) bind tuple identifiers (e.g. s_i is a tuple in the extension of r_i wrt SV_{k_i}). Clauses from (4) to (7) rule the attribute expressions appearing in the **`select`** clause. In particular, clauses in (4) bind attribute identifiers (e.g. a_i is the attribute named A_i in SV_{j_i}). Clauses in (5) verify that such attributes were in the schema of the relation

they belong to in the schema version where they have to be referenced (e.g. a_i is an attribute of S_{p_i} in SV_{j_i}). Clauses in (6) bind atomic values which are to be retrieved as the values of the desired attributes in the selected tuples (e.g. v_i is the value of a_i in the tuple identified by s_{p_i} wrt $SV_{k_{p_i}}$). In a similar way, clauses from (7) to (9) rule the attribute expressions appearing in the **`where`** clause. Finally, the F predicate in (10) enforces the selection conditions present in the **`where`** clause.

Now we relax the restriction on attribute expressions and consider the most general form of a multi-schema SPJ query, which can be expressed as the following MSQL **`select`** statement:

$$\begin{array}{l}\texttt{select SV}_{i_1}\texttt{:S}_{p_1}\texttt{.[SV}_{j_1}\texttt{:A}_1\texttt{], ..., SV}_{i_m}\texttt{:S}_{p_m}\texttt{.[SV}_{j_m}\texttt{:A}_m\texttt{]} \\ \quad\texttt{from SV}_{k_1}\texttt{:[SV}_{\ell_1}\texttt{:R}_1\texttt{] AS S}_1\texttt{, ..., SV}_{k_n}\texttt{:[SV}_{\ell_n}\texttt{:R}_n\texttt{] AS S}_n \\ \quad\quad\texttt{where } F(\texttt{SV}_{i'_1}\texttt{:S}_{p'_1}\texttt{.[SV}_{j'_1}\texttt{:A}_{q_1}\texttt{]}, \ldots, \texttt{SV}_{i'_r}\texttt{:S}_{p'_r}\texttt{.[SV}_{j'_r}\texttt{:A}_{q_r}\texttt{]})\end{array} \qquad (Q_{10})$$

Use of aliases and other notational conventions are the same as in query Q_9. The precise semantics in tuple calculus is:

$$\begin{aligned}\{\ (v_1,\ldots,v_m) \mid\ & \exists r_1 \cdots \exists r_n\ \exists s_1 \cdots \exists s_n\ \exists a_1 \cdots \exists a_m\ \exists a_{q_1} \cdots \exists a_{q_r}\ \exists v_{m+1} \cdots \exists v_{m+r} && (11)\\ & \mathsf{RName}_{\ell_1}(r_1, R_1) \wedge \ldots \wedge \mathsf{RName}_{\ell_n}(r_n, R_n) \wedge && (12)\\ & \mathsf{RExt}_{k_1}(r_1, s_1) \wedge \ldots \wedge \mathsf{RExt}_{k_n}(r_n, s_n) \wedge && (13)\\ & \mathsf{AName}_{j_1}(a_1, A_1) \wedge \ldots \wedge \mathsf{AName}_{j_m}(a_m, A_m) \wedge && (14)\\ & \mathsf{RSchema}_{j_1}(r_{p_1}, a_1) \wedge \ldots \wedge \mathsf{RSchema}_{j_m}(r_{p_m}, a_m) \wedge && (15)\\ & \mathsf{RExt}_{i_1}(r_{p_1}, s_{p_1}) \wedge \ldots \wedge \mathsf{RExt}_{i_m}(r_{p_m}, s_{p_m}) \wedge && (16)\\ & \mathsf{AExt}_{i_1}(s_{p_1}, a_1, v_1) \wedge \ldots \wedge \mathsf{AExt}_{i_m}(s_{p_m}, a_m, v_m) \wedge && (17)\\ & \mathsf{AName}_{j'_1}(a_{q_1}, A_{q_1}) \wedge \ldots \wedge \mathsf{AName}_{j'_r}(a_{q_r}, A_{q_r}) \wedge && (18)\\ & \mathsf{RSchema}_{j'_1}(r_{p'_1}, a_{q_1}) \wedge \ldots \wedge \mathsf{RSchema}_{j'_r}(r_{p'_r}, a_{q_r}) \wedge && (19)\\ & \mathsf{RExt}_{i'_1}(r_{p'_1}, s_{p'_1}) \wedge \ldots \wedge \mathsf{RExt}_{i'_r}(r_{p'_r}, s_{p'_r}) \wedge && (20)\\ & \mathsf{AExt}_{i'_1}(s_{p'_1}, a_{q_1}, v_{m+1}) \wedge \ldots \wedge \mathsf{AExt}_{i'_r}(s_{p'_r}, a_{q_r}, v_{m+r}) \wedge && (21)\\ & F(v_{m+1},\ldots,v_{m+r})\ \} && (22)\end{aligned}$$

As to the variables in (11), r_i, a_j, s_k, v_ℓ denote, respectively, a relation identifier, an attribute identifier, a tuple identifier and an atomic value. Clauses in (12) bind relation identifiers (e.g. r_i is the relation named R_i in SV_{ℓ_i}). Clauses in (13) bind tuple identifiers (e.g. s_i is a tuple in the extension of r_i wrt SV_{k_i}). Clauses from (14) to (17) rule the attribute expressions appearing in the **`select`** clause. In particular, clauses in (14) bind attribute identifiers (e.g. a_i is the attribute named A_i in SV_{j_i}). Clauses in (15) verify that such attributes were in the schema of the relation they belong to in the schema version where they have to be referenced (e.g. a_i is an attribute or S_{p_i} in SV_{j_i}). Clauses in (17) bind atomic values which are to be retrieved as the values of the desired attributes in the selected tuples (e.g. v_i is the value of a_i in the tuple identified by s_{p_i} wrt SV_{i_i}. Clauses in (16) verify that such tuples (in the r_{p_i} extension wrt SV_{k_i} by (13)) are also in the relation extension wrt SV_{i_i}. In a similar way, clauses from (18) to (21) rule the attribute expressions appearing in the **`where`** clause. Finally, the F predicate in (22) enforces the selection conditions present in the **`where`** clause.

For other query constructs, like grouping and aggregates, their standard semantics can be extended in a similar way when adding the new syntactic constructs (extensional and naming qualifiers).

4.1 Query Examples

Due to space limitations, we just give a couple of MSQL query samples with reference to our running example. We hope this can anyway give a flavor of the great flexibility and expressiveness that MSQL and the multi-pool approach can provide.

If we want to select the American names of all the cars also sold in Italy and in the rest of Europe, for which the Italian price is lower than the price applied in the rest of Europe, we can use the following query:

```
set schema USMKT;
select NAME
   from EUMKT:CAR AS EC, ITMKT:CAR AS IC
   where IC.PRICE < EC.PRICE
```

The **set schema** statement is used to select a default schema version [4, 2]. The same query could also be expressed regardless of the currently selected schema version as follows:

```
select USMKT:[USMKT:NAME]
   from EUMKT:[USMKT:CAR] AS EC, ITMKT:[USMKT:CAR] AS IC
   where IC.[USMKT:PRICE] < EC.[USMKT:PRICE]
```

If we want to assign to the "Lark 2.5" model sold in the US the same price at which it is sold in Italy, we can execute the following statement (assuming an exchange rate equal to 0.88):

```
set schema USMKT;
update CAR
   set PRICE = 0.88*ITMKT:PRICE
      where NAME = 'Lark 2.5'
```

If we only remember that such a car is called "Lark GT" in Italy (and we do not remember its American name), we have to write:

```
update CAR
   set PRICE = 0.88*ITMKT:PRICE
      where ITMKT:NAME = 'Lark GT'
```

The same query, written by an Italian programmer (aware of the names used in **ITMKT**) could probably be:

```
set schema ITMKT;
update USMKT:AUTO
   set PREZZO = 0.88*PREZZO
      where NOME = 'Lark GT'
```

Last but not least, we emphasize how the multi-schema mechanism can impact on the application development process. Consider the "localization" step of our example, which involved the translation of relation and attribute names into Italian in schema version **ITMKT**. Let us assume we want to select all the car models that are sold in Italy with a price lower than 25K Euros, but we ignore the names that have been given to **CAR** and **PRICE** in **ITMKT** (e.g. we only know the names in **USMKT**). Using MSQL we can easily "adapt" the query:

```
select * from CAR
   where PRICE < 25K
```

which was written to work on American data, as follows:

```
select * from ITMKT:[USMKT:CAR]
   where [USMKT:PRICE] < 25K
```

In other words, MSQL allows programmers to develop applications working on different schema versions, even if they do not know the names that have been assigned to tables and attributes on such versions. The query above works whichever such names are and even if names have not been changed at all. This adds great flexibility and availability to the application design process and allows a straightforward reuse for other schema versions of the software originally developed for any given schema version.

Finally notice that the use of surrogates to identify objects enables advanced features which are usually only available in an object-oriented database via the use of OIDs, and which can be best appreciated by applications thanks to schema versioning. For instance, such features include *object polymorphism* (which is inherent to our requirement that the same objects can have multiple (and possibly inconsistent) representations across data pools and *object migration*, as the same objects may have moved from a relation R_1 in a schema version SV_1 to another relation R_2 in schema version SV_2 (just the fact that the object is an instance of R_2 in SV_2 but it is also still an instance of R_1 in SV_1 allows us to detect that it has been migrated).

5 Conclusions

The main purpose of this paper was to present the MSQL multi-schema query language, which can be used in a relational database supporting schema versioning based on the multi-pool approach. MSQL extends SQL as it allows casual users and application developers to express multi-schema queries and enjoy the full potentialities of schema versioning. We also presented a simple logical storage model which can be used to implement schema versions and their data pools on top of a relational database. By means of this data model, the semantics of MSQL SPJ queries has been defined, and samples of how some basic operations of our model can be mapped on sequences of standard SQL operations have been provided in the Appendix.

References

1. Cristina De Castro, Fabio Grandi, and Maria Rita Scalas. On Schema Versioning in Temporal Databases. In James Clifford and Alexander Tuzhilin, editors, *Recent Advances in Temporal Databases – Proc. Intl' Workshop on Temporal Databases*, pages 272–291. Springer-Verlag, Berlin, 1995. Workshops in Computing.
2. Cristina De Castro, Fabio Grandi, and Maria Rita Scalas. Schema Versioning for Multitemporal Relational Databases. *Information Systems*, 22(5):249–290, 1997.
3. Klaus R. Dittrich and Raymond A. Lorie. Version Support for Engineering Database Systems. *IEEE Transactions on Software Engineering*, 14(4):429–436, 1988.
4. Richard T. Snodgrass (editor), Ilsoo Ahn, Gad Ariav, Don Batory, James Clifford, Curtis E. Dyreson, Ramez Elmasri, Fabio Grandi, Christian S. Jensen, Wolfgang Käfer, Nick Kline, Krishna Kulkarni, Ting Y. Cliff Leung, Nikos Lorentzos, John F. Roddick, Arie Segev, Michael D. Soo, and Suryanarayana M. Sripada. *The TSQL2 Temporal Query Language*. Kluwer Academic Publishing, New York, 1995.

5. Christian S. Jensen, Curtis E. Dyreson (eds.), Michael Böhlen, James Clifford, Ramez A. Elmasri, Shashi K. Gadia, Fabio Grandi, Pat Hayes, Sushil K. Jajodia, Wolfgang Käfer, Nick Kline, Nikos Lorentzos, Yannis Mitsoupoulos, Angelo Montanari, Daniel Nonen, Elisa Peressi, Barbara Pernici, John F. Roddick, Nandlal L. Sarda, Maria Rita Scalas, Arie Segev, Richard T. Snodgrass, Michael D. Soo, Abdullah U. Tansel, Paolo Tiberio, and Gio Wiederhold. The Consensus Glossary of Temporal Database Concepts - February 1998 Version. In Opher Etzion, Sushil Jajodia, and Suryanarayana Sripada, editors, *Temporal Databases — Research and Practice*, pages 367–405. Springer-Verlag, 1998. LNCS No. 1399.
6. John F. Roddick. A Survey of Schema Versioning Issues for Database Systems. *Information and Software Technology*, 37(7):383–393, 1995.
7. John F. Roddick and Richard T. Snodgrass. *Schema Versioning Support*, chapter 22, pages 427–449. In [4], 1995.
8. Randy H. Katz. Toward a Unified Framework for Version Modeling in Engineering Databases. *ACM Computing Surveys*, 22(4):375–408, 1990.
9. John F. Roddick. Implementing Schema Evolution in Relational Database Systems: An Approach Based on Historical Schemata. Tech. Rep. 10/93, Dept. of Computer Science and Computer Engineering, La Trobe University, October 1993.

A Operation Mappings

In this Appendix we show how some relevant operations can be effected via standard SQL, on top of a relation DBMS where schema versions and data pools have been implemented according to the logical storage model described in Sec. 3. This can be used as a simple (non optimized) implementation guideline for fast prototyping of a system based on our multi-schema relational model. We will use lower-case characters for MSQL statements and upper-case characters for their translation into standard SQL. We assume new surrogate values can be generated by referencing the system variable **new** [4].

Such definition are, in fact, specifications of the operational semantics of the involved statements. Although we gave here only some examples of SQL equivalent operations, for the sake of brevity and simplicity, a formal semantics of all the MSQL statements in their general form could also be provided as we did for the SPJ **select** in Sec. 4.

A.1 Schema Changes

Creation of a MSQL table in SV_i:

```
create table SVi:R(A,B)
```

can be translated into SQL as:

```
INSERT INTO RNamei VALUES (NEW,'R');
INSERT INTO ANamei VALUES (NEW,'A');
INSERT INTO ANamei VALUES (NEW,'B');
INSERT INTO RSchemai
   SELECT RID, AID FROM RNamei, ANamei
   WHERE RNamei.Name ='R' AND ANamei.Name IN ('A','B')
      AND AID NOT IN ( SELECT AID FROM RSchemai );
```

Renaming of a MSQL table in SV_i:

```
alter table SV_i:R rename as S
```

can be effected as:

```
UPDATE RName_i
SET RName_i.Name ='S' where RName_i.Name ='R'
```

Renaming of a MSQL table column in SV_i:

```
alter table SV_i:R alter column B rename as C
```

can be effected with SQL as:

```
UPDATE AName_i
SET AName_i.Name ='C'
   WHERE AName_i='B' AND AID =
      ( SELECT AID FROM RName_i, RSchema_i
       WHERE RName_i.Name ='R' AND RName_i.RID =RSchema_i.RID)
```

Creation of a new MSQL table column in SV_i:

```
alter table SV_i:R add column D
```

can be translated into SQL as:

```
INSERT INTO AName_i VALUES (NEW,'D');
INSERT INTO RSchema_i
   SELECT RID, AID FROM RName_i, AName_i
   WHERE RName_i.Name ='R' AND AName_i.Name ='D'
      AND AID NOT IN ( SELECT AID FROM RSchema_i );
INSERT INTO AExt_i
   SELECT TID, RSchema.AID, NULL
   FROM RName_i, AName_i, RSchema_i, RExt_i
   WHERE RName_i.Name ='R' AND AName_i.Name ='D'
      AND RSchema_i.RID =RName_i.RID AND RSchema_i.AID =AName_i.AID
      AND RExt_i.RID =RName_i.RID
```

Similar definitions can be given for all the relevant schema changes, including deletions, also if based on the "deactivation/reactivation" mechanism that has been proposed in the context of schema versioning under the single-pool approach (e.g. [9, 4]).

A.2 Modification Operations

Insertion of a tuple in a MSQL table in SV_i:

```
insert into SV_i:R(A,B) values ('a','b')
```

can be effected as:

```
INSERT INTO RExt_i
   SELECT RID, NEW
   FROM RName_i WHERE RName_i.Name ='R' ;
```

```
SELECT RID, TID INTO Temp
   FROM RExt_i WHERE TID NOT IN
      ( SELECT TID FROM AExt_i ) ;
INSERT INTO AExt_i
   SELECT TID, RSchema.AID, 'a'
   FROM Temp, AName_i, RSchema_i
   WHERE AName_i.Name ='A'
      AND RSchema_i.RID =Temp.RID AND RSchema_i.AID =AName_i.AID;
INSERT INTO AExt_i
   SELECT TID, RSchema.AID, 'b'
   FROM Temp, AName_i, RSchema_i
   WHERE AName_i.Name ='B'
      AND RSchema_i.RID =Temp.RID AND RSchema_i.AID =AName_i.AID;
```

Similar translations can be given also for the **update** and **delete** statements. Notice that, since they can be equipped with a **where** clause similar to the one used in the **select** statement, their translation requires a mix of code similar to the one listed above for the **insert**, plus some corresponding to the semantics of the **select** statements as to selection of tuples to be deleted/updated and selection of attribute values required to evaluate the **where** clause itself.

A.3 Enforcement of Constraints

For instance, we can show how the enforcement of a MSQL primary key constraint can be expressed with SQL on our logical storage model. Assume we want to check that (A_1, A_2, A_3) is the primary key of R in SV_i (for simplicity, we further assume that their respective identifiers, $\alpha_1, \alpha_2, \alpha_3$ and ρ, have already been determined). The following query will find all duplicate key values, if they exist:

```
SELECT A1.Value, A2.Value, A3.Value
FROM RExt_i, AExt_i AS A1, AExt_i AS A2, AExt_i AS A3
WHERE RExt_i.RID ='ρ'
   AND RExt_i.TID =A1.TID AND A1.AID ='α1'
   AND RExt_i.TID =A2.TID AND A2.AID ='α2'
   AND RExt_i.TID =A3.TID AND A3.AID ='α3'
GROUP BY A1.Value, A2.Value, A3.Value
HAVING COUNT(RExt_i.TID)>1
```

Similar checks can easily be embedded in assertions or triggers to enforce key constraints, and also other schema constraints (referential integrity, uniqueness, nulls, etc.). Obviously, controls can directly be added in the translation of modification operations (e.g. to reject the insertion of a new tuple if a tuple with the same key is already present).

This piece of SQL code is also the basic brick by means of which the schema changes involving the key definition (e.g. change of the primary key in a table, promotion/demotion of an attribute to/from the key) can be quite easily implemented in our model, whereas they are not so easy to be dealt with in the single-pool approach.